Macro Assembler Version 2.00

Programming Family

IBM

Personal
Computer
Software

6361612

Macro Assembler
Version 2.00

Programming Family

Personal
Computer
Software

Second Edition (August 1984)

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: International Business Machines Corporation provides this manual "as is," without warranty of any kind, either expressed or implied, including, but not limited to, the particular purpose. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this manual at any time.

This product could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication.

It is possible that this material may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Products are not stocked at the address below. Requests for copies of this product and for technical information about the system should be made to your authorized IBM Personal Computer dealer.

The following paragraph applies only to the United States and Puerto Rico: A Reader's Comment Form is provided at the back of this publication. If the form has been removed, address comments to: IBM Corp., Personal Computer, P.O. Box 1328-C, Boca Raton, Florida 33432. IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligations whatever.

© Copyright International Business Machines Corporation 1981, 1984

About This Book

This manual and the *Macro Assembler Reference* manual is for experienced assembler language programmers, who use (or intend to use) the IBM Personal Computer Macro Assembler or Small Assembler.

Specific information is provided on how to use the:

- Macro Assembler (MASM)
- Small Assembler (ASM)
- Structured Assembly Language Utilities (SALUT)
- Cross Reference Utility (CREF)
- Library Manager (LIB)
- Linker (LINK)

In addition to the above utilities, the architecture and control of the 8087, 8088, and 80286 is described for experienced programmers.

This Book is Organized as Follows

Chapter 1 "Introduction" introduces the Macro Assembler and its features. It also shows the application development process, and describes the utilities and files used in that process.

Chapter 2 "Assembly Language Format" explains assembler language statements, data representation, register usage, and other information you should understand to use the Macro Assembler effectively.

Chapter 3 "Memory and Segmentation" explains memory addressing and segment usage. It also describes the differences between .EXE and .COM files.

Chapter 4 "SALUT" describes The Structured Assembly Language Utilities functions that aid you in developing and listing your source programs.

Chapter 5 "Macro Assembler" describes the Macro Assembler, and the commands required to assemble your program. It also gives a brief overview of the remaining processes through program execution and debugging.

Chapter 6 "Cross Reference Utility (CREF)" describes the facility that aids you in debugging and listing your programs.

Chapter 7 "The Library Manager" describes the commands required to build and use a library.

Chapter 8 "The Linker (LINK) Utility" describes the Linker's functions and the commands required to link your program.

Chapter 9 "80286 Based IBM Personal Computers" describes the 80286 modes of operation, base architecture, memory access differences, and 80287 math co-processor.

Chapter 10 "IBM Personal Computer 8087 Math Co-Processor" describes the 8087 architecture, data types and formats, and processor control.

Appendix A "Messages" lists the messages produced by the assembler and its related utilities: LIB, LINK, and SALUT. I/O handler messages are also included.

Appendix B "Instruction Set Summary" summarizes the assembler instructions.

Appendix C "Pseudo Operations Summary" lists (in formatted form) the data, conditional, Macro, and listing pseudo operations.

Appendix D "The Sample Sessions" describes the requirements and procedures for using the sample source programs included on the diskette.

Glossary "Glossary" defines terms you might come across in this manual.

Conventions Used In This Book

Certain conventions are used in this manual to define commands, statements, functions, and variables:

1. Words in CAPITAL LETTERS are keywords and must be entered as shown. You can enter either uppercase or lowercase characters.

2. You supply any items in lowercase *italic letters*.

3. Items in square brackets ([ ]) are optional.

   Note: Do not confuse this convention with the square bracket notation used in the "Data Transferring Operands" section of Chapter 2 "Assembly Language Format".

4. An ellipsis (...) indicates an item can be repeated as many times as you wish.

5. All punctuation except square brackets, such as commas, parentheses, angle brackets, slashes, or semicolons must be included where shown.

6. "Macro Assembler" is the large assembler that supports all of the instructions and pseudo-ops described in this book.

7. "Small Assembler" is the small assembler that requires less memory but does not support all of the instructions and pseudo-ops described in this book.

8. The term, "assembler," is used to refer to both the Macro Assembler and the Small Assembler.

Related Books

IBM Personal Computer Disk Operating System

IBM Personal Computer Technical Reference
We recommend version 2.02 or later for 8087 users.

Reference Material

IBM Personal Computer *Macro Assembler Reference* version 2.0, included with this product.

Rector, Russell and George Alexy, *The 8086 Book, includes the 8088,* Osborne/McGraw-Hill, Berkeley, California, 1980.

The iAPX 86,88 User's Manual 210201, *The 8086 Family User's Manual* 9800722, Literature Department, Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA. 95051.

For 8087 users; *The 8086 Family User's Manual Numerics Supplement* 121586, Literature Department, Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA. 95051.

For 80286 users; *iAPX 286 Programmer's Reference Manual including the iAPX 286 Numeric Supplement* 210498, Literature Department, Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA. 95051.

Intel is a registered trademark of Intel Corporation
Microsoft is a trademark of Microsoft Corporation

Contents

Chapter 1. Introduction 1-1
Version 2.0 Enhancements 1-3
Diskette Contents 1-10
Assembler Diskette 1-11
Back Up the Assembler Diskette 1-14
General Information 1-15
The Application Development Process 1-16
Default Filename Extensions 1-17
Development Process 1-19
Creating Assembler Source Files 1-23
EDLIN Program 1-23
Personal Editor Program 1-24

Chapter 2. Assembly Language Format 2-1
Statements - Basic Units of Assembler Programs 2-3
The Name Field 2-3
The Operation Field 2-9
The Operand Field 2-10
The Comment Field 2-32
Construction Rules 2-33
Statement Format 2-38
Register Code Conventions 2-39
General Registers 2-39
Pointer (Offset) Registers 2-39
Segment Registers 2-41
Flags Register 2-42
Implicit Use of Registers 2-43

Chapter 3. Memory and Segmentation 3-1
Addressing Memory 3-3
Program Structure 3-6
.EXE File Structure 3-7
.COM File Structure 3-17
.EXE and .COM File Comparison 3-21

Chapter 4. Structured Assembly Language Utilities (SALUT) **4-1**
Introduction 4-3
Structure Statements 4-4
Structure Statement Syntax 4-6
Logic Flow Structures 4-9
How To Use SALUT 4-20
Starting SALUT 4-32
Error Messages 4-38

Chapter 5. Macro Assembler **5-1**
Introduction 5-3
Pass 1 and Pass 2 Differences 5-3
Phase Errors 5-5
Forward Reference 5-6
Device Identifications 5-6
The Assembly Step 5-8
Before You Begin 5-8
Parameters (/parms) 5-10
Option 1 — Console Responses 5-12
Option 2 — Command Line Responses 5-14
Option 3 — Batch Command 5-17
Assembler Termination 5-17
Viewing Assembler Files 5-18
Create a Cross Reference 5-19
Eliminate All Errors 5-20
The Linker Step 5-20
Viewing Linker Files 5-21
Executing Your Program 5-22
Use DEBUG 5-23
1. Reversed Order of Bytes 5-23
2. IP or SEG Displacement 5-24
3. Fields Modified by LINK or LOADER 5-25
Professional Debug Facility 5-25

Chapter 6. Cross Reference Utility (CREF) **6-1**
Introduction 6-3
Create a Symbol File 6-3
Before You Begin 6-4
How to Start CREF 6-4
Option 1 — Console Responses 6-4
Option 2 — Command Line Responses 6-6

CREF Termination 6-7
CRF Format 6-8

Chapter 7. The Library Manager (LIB) 7-1
Introduction 7-3
Terms 7-4
Library Manager Functions 7-4
Using the Library Manager 7-5
Library Operators 7-7
Entering Commands 7-8

Chapter 8. The Linker (LINK) Utility 8-1
Introduction 8-3
Files 8-4
Input Files 8-4
Output Files 8-4
VM.TMP (Temporary File) 8-5
Definitions 8-5
Segment 8-6
Group 8-6
Class 8-7
Command Prompts 8-7
Detailed Descriptions of the Command Prompts 8-9
Object Modules [.OBJ]: 8-9
Run File [filename.EXE]: 8-10
List File [NUL.MAP]: 8-11
Libraries [.LIB]: 8-12
Linker Parameters 8-14
Starting the linker Program 8-17
Before You Begin 8-17
Option 1 — Console Responses 8-17
Option 2 — Command Line Responses 8-18
Option 3 — Automatic Responses . . . 8-20
Example linker Session 8-22
Determining the Absolute Address of a Segment 8-25
Messages 8-26

Chapter 9. 80286-Based IBM Personal Computers 9-1
Introduction 9-3
Modes of Operation 9-3
Base Architecture 9-4

Memory Access Differences 9-5
80287 Math Co-processor 9-6

Chapter 10. IBM Personal Computer 8087 Math Co-Processor 10-1
Introduction 10-3
8087 Architecture 10-5
Environment Data 10-5
Save State 10-6
Exception Pointers 10-8
Register Stack 10-9
Status Word 10-11
Control Word 10-15
Exceptions 10-18
Tag Word 10-21
Data Types 10-22
Selecting Data Types 10-23
Data Formats 10-24
Binary Integers 10-24
Decimal Integers 10-26
Real Numbers 10-27
Special Values 10-31
Data Format Requirements 10-33
Data Format Conversion 10-37
Data Conversion Routines 10-38
Processor Control and Monitoring 10-45
Co-processing 10-45
Initialization 10-45
Interrupts and Waits 10-47

Appendix A. Messages A-3
Assembler Messages A-3
Small Assembler (ASM) A-3
Macro Assembler (MASM) A-3
I/O Handler Messages A-12
Runtime Messages A-14
Library Manager Error Messages A-15
Unrecoverable Errors A-15
Other Errors A-17
LINK Messages A-18
SALUT Error Messages A-24

Appendix B. Appendix B. Instruction Set Summary B-1

8088 Instruction Set Summary B-1
8087 Instruction Set Summary B-16
80286 Instruction Set Summary B-27

Appendix C. Pseudo Operations Summary C-1
Conditional Pseudo Operations C-2
Data Pseudo Operations C-3
Listing Pseudo Operations C-5
Macro Pseudo Operations C-6
    Special Macro operators C-6
Mode Pseudo Operations C-6

Appendix D. The Sample Sessions D-1
Introduction D-1
    Requirements D-3
TRYCOM Sample Sessions D-4
    Small Assembler Session D-5
    Macro Assembler Session D-10
SALUT Sample Session D-12
Assembler Source Coding Aids D-23

Glossary Glossary-1

Index Index-1

Chapter 1. Introduction

Contents

Version 2.0 Enhancements **1-3**
- Technical Differences 1-3
- Functional Enhancements 1-6
- Increase in Memory Requirements 1-7
- 80286 Processor 1-8
- 8087/80287 Numeric Processor 1-8
- Pathname Support 1-8
- LIB Utility 1-9
- LINK Utility 1-9
- SALUT Utility 1-9
- The Sample Sessions 1-9

Diskette Contents **1-10**
- Assembler Diskette 1-11
- Back Up the Assembler Diskette 1-14

General Information **1-15**

The Application Development Process **1-16**
- Default Filename Extensions 1-17
- Development Process 1-19

Creating Assembler Source Files **1-23**
- EDLIN Program 1-23
- Personal Editor Program 1-24

Version 2.0 Enhancements

The Macro Assembler version 2.0 contains significant functional enhancements and some minor operational and technical differences from the previous version.

Technical Differences

Error checking is improved. Source that assembled without errors in version 1.0 may produce errors when assembled with version 2.0. Some errors are the result of detecting errors that were not detected before, others show that the assembler is making an assumption and is warning the user that the specification was not complete. In version 1.0 some assumptions were made without warning the user.

Certain situations caused the version 1.0 assembler to generate Phase Errors. The Phase Errors were due to a failure of pass 1 to communicate completely with pass 2. This would cause pass 2 to add NOP instructions that were not needed in a mistaken effort to avoid Phase Errors.

The version 2.0 assembler is more TYPE sensitive. Certain TYPE mismatches in version 1.0 were ignored. These TYPE mismatches are now flagged as errors in version 2.0. These errors can be corrected by the addition of the proper explicit TYPE override, using the PTR operator. For example:

- If a variable name was defined with an ending colon, it becomes a NEAR label instead of a variable name. Referencing such a symbol as if it were a WORD was allowed in version 1.0, however version 2.0 detects this as a TYPE Mismatch error. Never use a colon on a variable name.

- In the following example POINTR DD SUBROUT sets up a double word vector to contain the offset and seg id (segment identification) of a subroutine. These two words of that vector could be referenced in version 1.0 as:

  MOV POINTR+2,AX

  However V2.0 produces a TYPE Mismatch since AX is TYPE=WORD and POINTR is TYPE=DWORD thus requiring a PTR override. The proper reference is:

  MOV WORD PTR POINTR+2,AX

For example:

```
POINTR DD SUBROUT
MOV POINTR+2,AX ;IMPROPER BUT ACCEPTED
                ; IN V1.0
                ;PRODUCES TYPE MISMATCH
                ; IN V2.0
MOV WORD PTR POINTR+2,AX ;PROPER CODING
```

The SHL, SHR, and XOR operators now work as documented. If your source was written to use these operators and debugging caused source to be altered to match the assembler (rather than the documentation), the source may need to be altered back to match the documentation. Possibly the two expressions around those operators will need to be swapped, but not necessarily, because for some values of those expressions the operators performed as documented.

Packed decimal (using the DT data definition pseudo-op) was in version 1.0 producing the bytes in backwards order.

Certain floating-point constants produced incorrect encoding. For example, compare the generated code for:

```
DD  -1048576.125
```

Nested segments in version 1.0 produced an invalid relocation counter, under certain circumstances, thus requiring that a segment be terminated before starting another. In version 2.0, it is no longer necessary to terminate a segment before defining an inner segment.

Control blocks defined using the STRUC can now be generated using DUP.

The size of the area defined using the STRUC has been enlarged. Size is not defined in number of bytes of memory being defined, but the size of the assembler's workspace needed to contain the definition of the STRUC. STRUC uses a mechanism similar to DUP when communicating to Linker, thus causing the curious message "DUP too large for Linker" when the work area limit for a STRUC is exceeded (when it was not obvious a DUP was involved).

The assembler still uses a 64K work area. Once the RAM size in a machine goes above what is needed to provide that 64K work area, additional RAM is ignored.

Using STRUC to generate DD vectors to subroutines in version 1.0 had a problem that required modification to both LINK and the assembler. If you have to use a level of LINK whose version is previous to the LINK version included on the assembler diskette, these vector pointers may not be properly initialized. The LINK included on the assembler diskette requires DOS 2.0 or later. If you are running DOS 1.1, be aware of that limitation of your LINK.

Functional Enhancements

- Return code (ERRORLEVEL) returned to Batch processor instead of setting a byte value in an undocumented (therefore unsupported) location in fixed low memory RAM.
- Ordering of segments may be either alphabetic (default, same as in version 1.0) or physical sequential, the option is selectable using a /parameter.
- Printing of a symbol table at the end of the listing can now be suppressed by specifying a /parameter.
- Performance has been significantly improved, especially for those modules that make heavy use of macros.

- The instruction set has been enlarged to include all the instructions for the 8087 Math co-processor (floating-point arithmetic) and for the subset of the 80286 processor that is compatible with the 8088 addressing mode. Be aware that previous support of the 8087 has been in the form of several macro libraries produced by a variety of non-IBM sources. Any assembler code written to use these libraries may not necessarily be following the same set of syntax rules required for the 8087 support of this new assembler and thus may need to be corrected.

  Note: DOS assemble and unassemble debug commands support 8087 instructions, but not 80286 instructions.

- Three source files (SKELEXE.ASM, SKELEXE2.ASM, and SKELCOM.ASM) are included on the diskette for an aid in coding new source programs. When you start to write an assembler application, use one of these "skeleton" source files as your model, adding your application's code into the indicated positions. These models illustrate the several modes of interfacing to DOS at entry and exit times.

Increase in Memory Requirements

Because of the significant amount of function added, the Macro (large) Assembler now requires at least 96K memory with DOS 1.1, or 128K with DOS 2.0. The Small Assembler now requires at least 96K memory with DOS 1.1 or DOS 2.0.

> **Note:** During the assembly process, the Macro Assembler uses up to 192K storage, if available (144K plus the storage required by DOS). The Small Assembler uses up to 115K.

80286 Processor

The Macro Assembler version 2.0 has 16 new instructions for 80286-based IBM Personal Computers.

8087/80287 Numeric Processor

The Macro Assembler version 2.0 supports The IBM Personal Computer 8087 and 80287 Math Co-Processors. The assembler has 69 new floating-point instructions for the 8087 and 80287.

The Macro Assembler version 2.0 allows 8087 emulation if used concurrently with the *IBM Personal Computer Fortran version 2.0*, or the *IBM Personal Computer Pascal version 2.0* products.

The Macro Assembler (MASM) uses the 8087 if installed, or emulates the 8087 if not present, during the assembly process of some floating-point define-variable operations.

> **Note:** The Macro Assembler may "hang" if an 8087 co-processor is installed but the system board switches are set for "8087 not installed". When the assembler attempts to assemble a floating-point define-variable operation, an infinite loop (endless wait) may occur. Set switch block 1, switch position 2 to "on" for no 8087 and "off" for 8087 installed. See your IBM Personal Computer *Guide to Operations* for details on switch settings and locations.

Pathname Support

The Macro Assembler version 2.0 allows the use of Pathnames with DOS 2.0 and later.

LIB Utility

New Utility Program

LIB (Library Manager) allows the user to develop libraries from object files. This program is described in Chapter 7, "The Library Manager."

LINK Utility

New Utility Program

A new, more powerful linker is now included with the Macro Assembler. This linker can only be used with DOS version 2.0 or later. This program is described in Chapter 8, "The Linker (LINK) Program."

SALUT Utility

New Utility Program

SALUT (The Structured Assembly Language Utilities) is a preprocessor designed to aid in creating and formatting structured assembler code. This program is described in Chapter 4, "Structured Assembly Language Utilities (SALUT)."

The Sample Sessions

Appendix D, "The Sample Sessions", has been expanded to demonstrate file structure, coding conventions, input/output routines, structured programming, and rules of syntax.

The sample programs included with the IBM Personal Computer Macro Assembler are highly tutorial. We recommend that all users of this product, regardless of their experience level, go through the sample sessions. Pay particular attention to the statement comments, program listings and session procedures.

See Appendix D, "The Sample Sessions," for details on using the sample source files and Chapter 5 "Macro Assembler" for details on using the assembler.

Diskette Contents

The diskette included with The IBM Personal Computer Macro Assembler is double-sided with eight sectors per track for compatibility with earlier versions of DOS. Because of the size, number, and format of the included files, the diskette is almost full. We recommend that you copy the diskette using the DOS COPY command (not DOS DISKCOPY) selecting the programs you wish to have on each of your work diskettes, or hard disk. By using this method, you can choose the format that best suits your level of DOS and the hardware available on your IBM Personal Computer.

Assembler Diskette

The IBM Personal Computer Macro Assembler Diskette contains these files:

ASM.EXE
The Small Assembler program.

CREF.EXE
The cross reference utility program that produces an alphabetic listing file of data, code and segment-relative symbols.

FLINS.ASM
An assembler source module demonstrating 8087 instructions. The resultant listing can be used as a floating-point reference and is not intended to be executed.

IBMASM.ASM
A source subroutine that interfaces with floating-point conversion modules residing in the library. It is called by UTEST.PAS (defined below) and is used by Pascal applications.

IBMUTIL.LIB
A group (library) of source modules that perform floating-point conversions.

LIB.EXE
The Library Manager utility—an aid in managing object files as members of a library.

LINK.EXE
The new version of the DOS 2.0 Linker (requires DOS 2.0 or later).

MASM.EXE
The Macro Assembler program.

SALEX.SAL

The main module of the SALUT sample session that is linked to "SALEX" files 1-4 and STACK.ASM. This sample session demonstrates the composition of a structured application that must be processed by the SALUT utility.

SALEX1.SAL

One of five "SALEX" modules that makes up the SALUT sample session.

SALEX2.SAL

One of five "SALEX" modules that makes up the SALUT sample session.

SALEX3.SAL

One of five "SALEX" modules that makes up the SALUT sample session.

SALEX4.SAL

One of five "SALEX" modules that makes up the SALUT sample session.

SALEXINC.ASM

An include file that contains an example of a structure. It is referenced by several SALEX modules (see SALEX.SAL definition below).

SALUT.EXE

The Structured Assembler Language Utilities—an aid to creating, printing and reading source files.

SAMPLE2.BAT

This sample batch file, used in the macro assembler sample session, demonstrates an alternate method of development process command entry.

SKELCOM.ASM

This sample source file is designed to demonstrate a .COM structured framework within which source code can be inserted.

SKELEXE.ASM

This sample source file is designed to demonstrate an .EXE structured framework within which source code can be inserted.

SKELEXE2.ASM

This sample source file also demonstrates an .EXE structured file, but is designed specifically for DOS version 2.0 (or later).

STACK.ASM

This source file is used in conjunction with the SALEX group (below) and demonstrates the definition of stacks.

TRYCOM1M.ASM

This sample source module 1 demonstrates .COM file structure, I/O routines and .COM module linkage. It is input to the Macro Assembler (MASM).

TRYCOM1S.ASM

This sample source module 1 demonstrates .COM file structure, I/O routines and .COM module linkage. It is input to the Small Assembler (ASM).

TRYCOM2M.ASM

This sample source module 2 is linked to TRYCOM1M and is input to the Macro Assembler (MASM).

TRYCOM2S.ASM

This sample source module 2 is linked to TRYCOM1S and is input to the Small Assembler (ASM).

UTEST.PAS

A Pascal (version 2.0) source module that interfaces with IBMASM.ASM (defined above), appropriate Pascal files from the Pascal library, and the floating-point conversion modules.

Back Up the Assembler Diskette

Back up your Macro Assembler diskette before you use it by following these procedures:

One Diskette Drive User:

- Format one blank diskette using the DOS FORMAT command, without the /S option.
- Insert Assembler Diskette into drive A and enter:

```
COPY A:*.* B:
```

- Respond to the DOS prompts.

Two Diskette Drive User:

- Format one blank diskette using the DOS FORMAT command, without the /S option, and insert the diskette into drive B.
- Insert Assembler Diskette into drive A and enter:

```
COPY A:*.* B:
```

Fixed Disk User:

- Set the DOS default drive to the fixed disk device (C, D, E, or F depending on your system's configuration).
- Enter the DOS MKDIR command to create a sub-directory for the assembler files.
- Enter the DOS CHDIR command to set the default sub-directory to the newly created assembler directory.
- Insert Assembler Diskette into drive A.

- Copy Assembler Diskette into the new sub-directory by using the following DOS command:

```
COPY A:*.*
```

Keep the original assembler diskette in a safe place and separate from the original copy. Use the copy for day-to-day operation.

General Information

The IBM Personal Computer Macro Assembler translates *source* code that you have coded in the assembler language into an *object* module containing the machine language understood by the computer.

You will be using either the Small Assembler (96K) or the Macro Assembler (128K). If you have 128K, you can use either assembler, but we recommend that you use the Macro Assembler which supports all the functions described in this manual. The Small Assembler does not support the 8087 and 80287 instructions and some pseudo-ops.

The Small Assembler identifies each error by displaying only an error number (identifier); the Macro Assembler identifies each error with an error number and a defining statement.

Both Assembler's run under the Disk Operating System described in the IBM Personal Computer *Disk Operating System (DOS)* manual.

The Application Development Process

Assembling source code is the first step of converting your source text into an application program. After assembly, you must link the application program before you begin its execution. Let us quickly review the key files that are created in this process.

Among the many files that can be created from your original text, three are of primary importance:

- Source file (.ASM)
- Object file (.OBJ)
- Execution file (.EXE or .COM)

Once you have written and identified your *source* files using a DOS-compatible editor, add the filename extension .ASM to the filename. For example:

```
NEWFILE.ASM
```

By adding this extension, you are identifying the file as an assembler source code file. If your source code contains Structured Statements (identified by the dollar sign ($)), your source file must be defined as, *filename*.SAL, and processed by the SALUT utility before using the assembler (see Chapter 4 "Structured Assembly Language UTilities" for details).

The assembler reads the source file and converts this file into an *object* file that contains the machine-readable object code. If errors occur in the file, go back to the source, edit, and reassemble the text until it is error-free.

When the assembler returns no error messages, the object file can be made into an .EXE file by the linker (see Chapter 8 "The Linker (LINK) Utility" for a detailed description of the LINK utility). Depending on the file's structure, you can either run the application as an .EXE file or convert the file—using the DOS EXE2BIN command—into a .COM file that can then be executed (see the DOS manual for a detailed description of the EXE2BIN utility).

Default Filename Extensions

The default filename extensions consist of a period and three characters (.XXX). An extension is a file identifier appended to a filename. For example:

```
MYFILE.ASM
```

"MYFILE" identifies the name of your file, while the extension ".ASM" informs the assembler that your file contains assembler source code.

This table defines the input files required by each utility and the output files produced by each utility. The utilities are listed in their respective order of usage.

SALUT Utility

Input:	*filename*.SAL Structured source file that must be processed by SALUT. Note: Renamed to, .BAK, by SALUT before processing.
Output:	*filename*.SAL Reformatted (according to specified parameters) source file listing. *filename*.ASM Structured Statements expanded to assembler source that can be processed by the [M]ASM utility.

[M]ASM Utility

Input:	*filename*.ASM Assembler source file produced by an editor or SALUT.
Output:	*filename*.OBJ Relocatable object file containing machine language code. *filename*.LST Printable assembler listing file. *filename*.CRF Symbol file to be processed by CREF.

CREF Utility

Input:	*filename*.CRF Symbol file produced by [M]ASM.
Output:	*filename*.REF Printable cross reference file.

LIB Utility

Input:	*filename*.LIB Original library file (may be modified). *filename*.OBJ Object files produced by the assembler (or a compiler).
Output:	*filename*.LIB User or compiler library file produced by the LIB utility. *filename.ext* (no default provided) Printable "list" file of all library members.

LINK Utility

Input:	*filename*.OBJ Object file produced by the assembler (or a compiler). *filename*.LIB User or compiler library file containing the required members.
Output:	*filename*.EXE Run file (may or may not be executable). *filename*.MAP Printable list file containing the memory map.

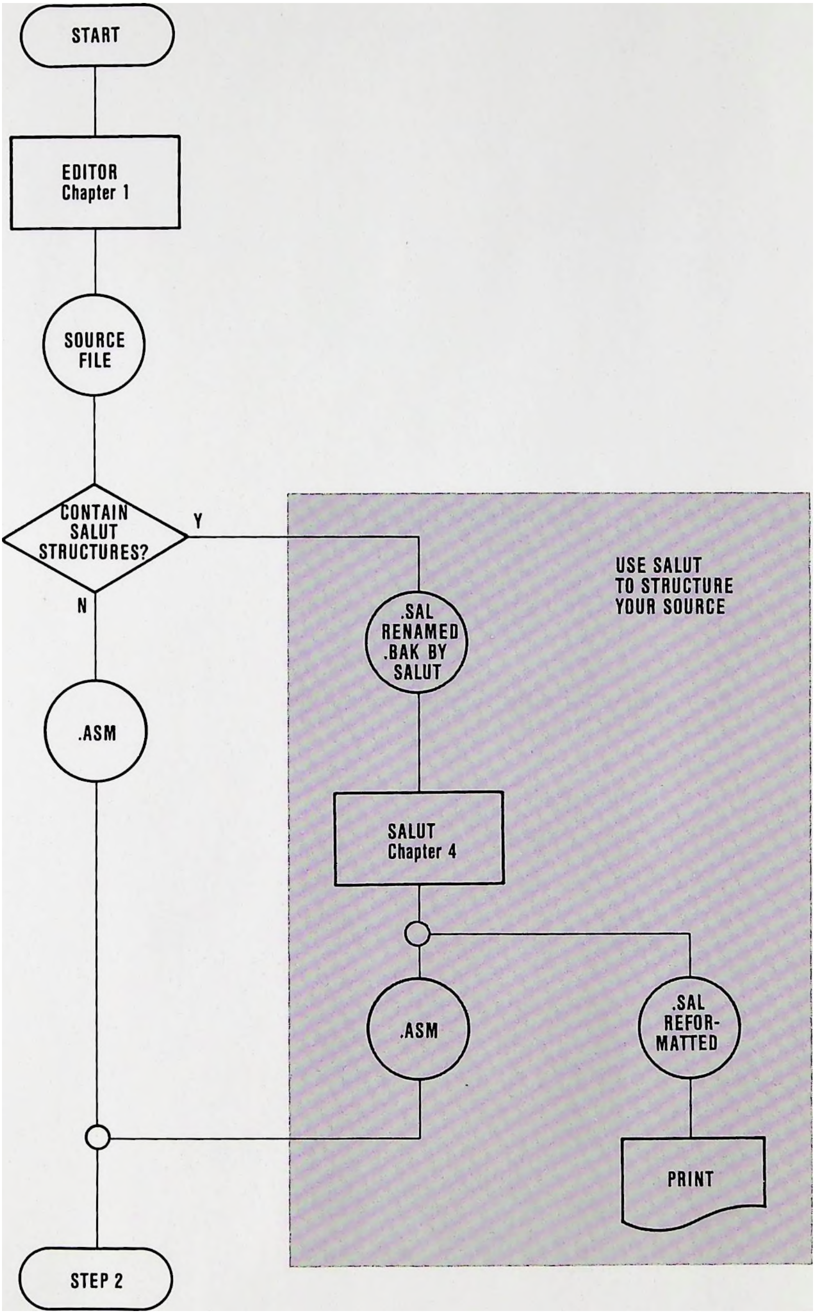
EXE2BIN Utility

Input:	*filename*.EXE Run file with one common segment.
Output:	*filename*.COM Executable, self-relocatable command file.

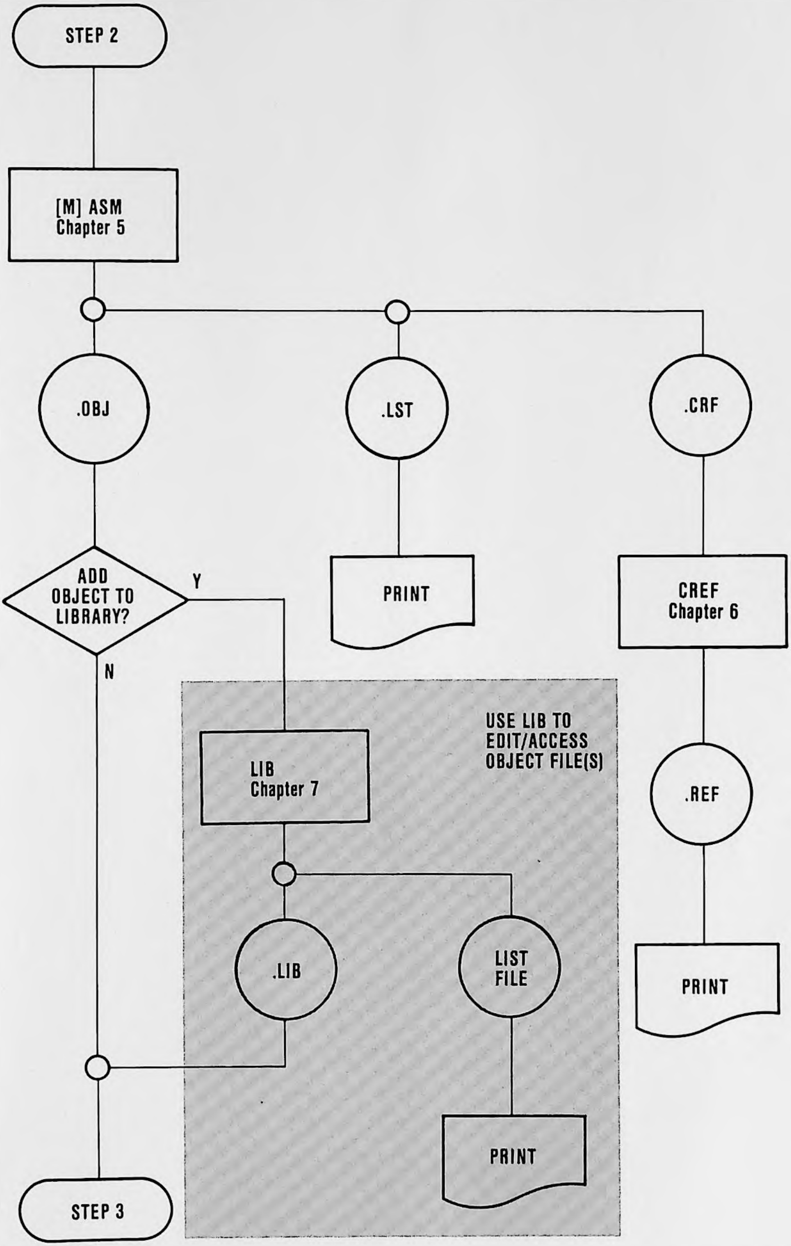
Development Process

The input/output files and their respective utilities combine to form the development process. The following flow charts break the development process into three steps with one step per chart.

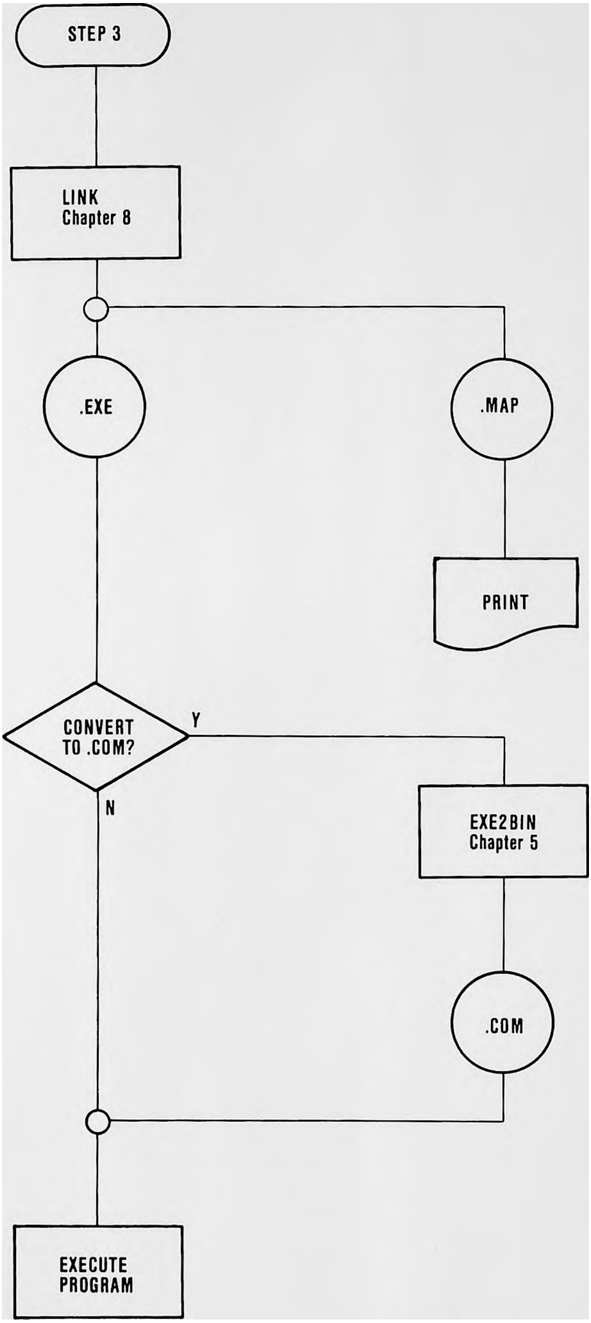
Step 1 – Create the Source


START
EDITOR
Chapter 1
SOURCE FILE
CONTAIN SALUT STRUCTURES?
Y
N
.ASM
USE SALUT TO STRUCTURE YOUR SOURCE
.SAL RENAMED .BAK BY SALUT
SALUT
Chapter 4
.ASM
.SAL REFOR-MATTED
PRINT
STEP 2

Step 2 - Assemble the Source (Produce the Object)


STEP 2
[M] ASM
Chapter 5
.OBJ
.LST
.CRF
ADD
OBJECT TO
LIBRARY?
Y
N
PRINT
CREF
Chapter 6
USE LIB TO
EDIT/ACCESS
OBJECT FILE(S)
LIB
Chapter 7
.REF
.LIB
LIST
FILE
PRINT
PRINT
STEP 3

Step 3 - Link the Objects, Produce the Run File


STEP 3
LINK
Chapter 8
.EXE
.MAP
PRINT
CONVERT TO .COM?
Y
N
EXE2BIN
Chapter 5
.COM
EXECUTE PROGRAM

Creating Assembler Source Files

To create source code, you must have a means of writing, formatting and filing your text. The Line Editor (EDLIN) and The Personal Editor were designed to carry out these tasks.

> **Note:** Like the Macro Assembler, EDLIN and Personal Editor are DOS compatible, that is, you enter either of these programs after first entering the DOS mode. These editors are not, however, built into the Macro Assembler.

EDLIN Program

The EDLIN program is a text editor you can use to create, alter, and display source files input to the assembler. EDLIN is part of the IBM Personal Computer *Disk Operating System (DOS)* package.

This list describes the EDLIN features:

- Deletes, edits, inserts, and displays lines
- Copies and moves blocks of text (DOS version 2.0 and 2.1)
- Searches for, deletes, replaces, and displays text
- Creates new files and saves them
- Updates old files and saves both updated and original files.

See the DOS manual for a detailed description of the EDLIN program.

Personal Editor Program

The IBM Personal Computer "Personal Editor" program (available as a separate software package) can also be used to create, change and display files. Unlike EDLIN, however, Personal Editor is a full-screen editor that allows you to make on-screen changes without regard to lines or line numbers.

In addition to full-screen editing, the Personal Editor provides:

- Paragraph, line and sentence markers to let you move, copy, erase and overlay text
- Word processing features like tab and margin setting, word wrapping and reformatting of text
- Editing of up to 20 DOS ASCII files at one time in memory
- Ability to redefine 99 keys on the keyboard
- Help Menu
- Block Shifting
- Diskette Directory display.

Chapter 2. Assembly Language Format

Contents

Statements - Basic Units of Assembler Programs . 2-3
The Name Field 2-3
Data Relative Items — Variables 2-4
Code Relative Items — Labels 2-4
Absolute Items — Constants and Equates 2-6
Segment Names 2-8
The Operation Field 2-9
Machine Instructions 2-9
Macro Instructions 2-9
Pseudo Operations 2-9
The Operand Field 2-10
Data-Transferring Operands 2-10
Coding Operands 2-14
Operators 2-18
The Comment Field 2-32
Construction Rules 2-33
Statement Format 2-38

Register Code Conventions 2-39
General Registers 2-39
Pointer (Offset) Registers 2-39
Base Pointers 2-40
Index Pointers 2-40
Stack Pointer 2-40
Instruction Pointer 2-40
Segment Registers 2-41
Flags Register 2-42
Status flags 2-42
Processor control flags 2-43
Implicit Use of Registers 2-43

Statements - Basic Units of Assembler Programs

The assembler program (source code text) is composed of a series of *statements*. Each statement is a line of text that instructs the assembler to carry out a task.

Assembler statements have four *fields* that must be written, from left to right, in this order:

Name Field	**Operation Field**	**Operand Field**	**Comment Field**

Depending on the purpose of the statement, some fields are optional and may be left blank. The comment field is completely optional (not required) and is provided for programming documentation only.

The Name Field

A *name* field entry is a symbol that is referenced to identify the memory location of an item. For example:

```
NUMBER1    DB    "1"
```

"NUMBER1" identifies the memory location of the byte that contains the character, "1".

Name field entries identify four kinds of items:

- Data relative items — Variables
- Code relative items — Labels
- Absolute items — Constants and Equates
- Segment names.

Data Relative Items — Variables

Data relative items are defined in the data section of the source file.

Variables

A *variable* is a data-defined area of memory that is accessible by the application at execution time.

Variables have five attributes associated with their usage: segment, offset, type, length, and size. The *segment* and *offset* attributes define the address of the variable in memory. (Addressing is described in Chapter 3 "Memory and Segmentation.") The *type* attribute defines the number of consecutive bytes reserved for a variable. The DB, DW, DD, DQ, DT, RECORD, and STRUC pseudo-ops are associated with variable types (see the *Macro Assembler Reference* book for details on each of these pseudo-ops).

Variables can be used as operands in simple, indexed, or structured forms (see "The Operand Field" in this chapter).

Code Relative Items — Labels

Unlike variables, *labels* are defined in the executable code section of an application. Because they are code-relative, labels can only be defined in the segment whose beginning address is contained in the Code Segment register, CS. the code segment. (See Chapter 3 "Memory and Segmentation" for details on the code segment.)

Labels also have three attributes associated with their usage: segment, offset, and type. The segment and offset attributes define the address of the label in memory. (Addressing is described in Chapter 3 "Memory and Segmentation".) The *type* attribute indicates whether the label can be referenced as NEAR or FAR.

When a label is used by itself (with no other code on the the same line) or in the name field of an instruction, it can be explicitly defined as type NEAR by a colon (:) or a pseudo-op. To define a label as NEAR with a colon, place the colon immediately after the label. For example:

```
ENTER:
```

A label can be explicitly defined as NEAR by using the LABEL pseudo-op, for example:

```
SYM  LABEL  NEAR
```

or the EQU pseudo-op, for example:

```
SYM  EQU  $
```

or the PROC pseudo-op, for example:

```
SYM  PROC  NEAR
```

or the EXTRN pseudo-op, for example:

```
EXTRN  SYM:NEAR
```

A label can be defined as FAR by using the LABEL pseudo-op, for example:

```
SYM  LABEL  FAR
```

or the EQU pseudo-op, for example:

```
SYM  EQU  FAR  PTR  $
```

or the PROC pseudo-op, for example:

```
SYM  PROC  FAR
```

or EXTRN pseudo-op, for example:

```
EXTRN  SYM:FAR
```

Whether a label is defined as type NEAR or type FAR is dependent on the construction of your application, that is, the application's modular construction, subroutine usage, and file type (.EXE or .COM). To gain a better understanding of how to assign NEAR and FAR type attributes, see Chapter 3 "Memory and Segmentation", the EXTRN, LABEL and PROC pseudo-ops in Chapter 2 of the *Macro Assembler Reference* book, and the SALUT and TRYCOM examples on the Assembler Diskette.

Absolute Items — Constants and Equates

Constants (Immediate Values)

Constants are numbers without attributes other than their immediate values. Constants can be given in these forms:

Binary: A sequence of 0's and 1's followed by the letter B. For example:

```
00101100B
```

Decimal: A sequence of digits from 0 through 9 with or without the letter D; unless the default is changed by the RADIX pseudo-op. (See the RADIX pseudo-op in the *Macro Assembler Reference* book). For example:

```
178 or 178D
```

Hexadecimal: A sequence of digits from 0 through 9 and letters A through F followed by the letter H. The first character must be one of the digits 0 through 9. For example:

```
OFFFH
```

Octal: A sequence of digits 0 through 7 followed either by the letter Q or the letter O. For example:

```
1777Q or 1777O
```

Character: All allowable characters are ASCII based and must be enclosed within single or double quotes. (See "Construction Rules" in this chapter for a list of the allowable characters.) If more than 2 characters are defined in one statement, the DEFINE BYTE (DB) pseudo-op must be used. If a character string is defined as 1 or 2 characters, the DD, DQ, DT and DW pseudo-ops can be used. (See the DEFINE DATA pseudo-ops in Chapter 2 of the *Macro Assembler Reference* book for details.) For example:

```
"C", 'C ALBURY'
```

Decimal Scientific Notation: Floating-point decimal digits. For example:

```
2.725E-1
```

> **Note:** Decimal Scientific Notation is not supported by the Small Assembler (ASM).

Hexadecimal Real: Hexadecimal real digits (0 through 9 and A through F) followed by the letter R. The first character must be one of the digits 0 through 9. The total number of digits must be 8, 16, or 20 unless the first digit is 0; then the total number of digits are 9, 17, and 21. For example:

```
OFFFFFFFFR
```

Note: Hexadecimal real notation is not supported by the Small Assembler (ASM).

Equates

A name field entry can be assigned the value of an operand field expression by using the EQU and EQUAL SIGN (=) pseudo-ops. For example:

```
HNRYLV EQU [BP+8]
TMBRNR  =  40
```

"HNRYLV" can now be used in place of the index expression "[BP+8]" and "TMBRNR" can be used in place of the immediate value "40". (See Chapter 2 of the *Macro Assembler Reference* book for details on the EQU and Equal Sign (=) pseudo-ops.)

Segment Names

Segments are labeled in the name field of the segment statement. For example:

```
STACKSEG   SEGMENT   PARA STACK 'STACK'
```

"STACKSEG" is the name field entry that the SEGMENT pseudo-op associates to the stack segment. (See Chapter 3 "Memory and Segmentation" and the SEGMENT pseudo-op in Chapter 2 of the *Macro Assembler Reference* book for details on segments.)

The Operation Field

An *operation* field entry is an instruction *mnemonic* that represents a machine, MACRO, or pseudo-op instruction. For example:

```
PUSHF
```

"PUSHF" tells the assembler to generate a machine instruction that stores the contents of the Flags register onto the stack segment.

Machine Instructions

The machine instructions are the symbolic representation of the hardware instructions in the assembler's instruction set.

Macro Instructions

The macro instructions are requests to the assembler program to process a predefined sequence of code. From this sequence, the assembler generates source instructions, which it then processes as if they were part of the original input in the source module.

Pseudo Operations

Pseudo operations, also called *pseudo-ops,* tell the assembler what to do with data, conditional branches, macros, and listings. Pseudo-ops usually *do not* produce machine language code. (See Chapter 2 "Pseudo Operations" in the *Macro Assembler Reference* book for a description of the pseudo-ops used in your Macro Assembler.)

The Operand Field

An *operand* is an entity to which an operation is applied. An operand, therefore, is the object of an operation mnemonic.

An operand field entry is an *expression* that contains at least one operand required by the operation mnemonic to carry out the statement's function. For example:

```
PUSH        DS
```

An operand (in this example "DS" is chosen) is required by the operation mnemonic "PUSH" to carry out its function. The complexity of an operand entry can vary according to the purpose of the statement. More complex expressions can contain a combination of two or more operands and/or *operator*s.

Data-Transferring Operands

Data-transferring operands are expressions required by certain operation mnemonics, such as, "MOV" or "ADD", to transfer data from one source to another. Data-transferring instructions usually require two operands (a source operand and a destination operand) to complete their operation. The assembler recognizes the left operand as the destination and the right operand as the source.

> **Note:** Use a comma (,) to separate destination and source.

A destination operand is the target location of a data transference. A source operand is the origin of the data. In this example:

```
MOV   AX,8[BX][SI]
```

the data represented by the operand to the right of the comma is transferred to the operand to the left of the comma.

Data-transferring operations allow either storage or register(s) to serve as the destination operand, and either storage, register(s), or constant(s) to serve as the source operand. (See Chapter 1 of the *Macro Assembler Reference* book for more details on source and destination operands.)

Unless data is classified as string data and transferred by a string instruction mnemonic, such as, MOVS, it cannot be transferred from memory to memory. (See the string instructions: MOVS, MOVSB, MOVSW, CMPS, CMPSB, AND CMPSW in Chapter 3 of the *Macro Assembler Reference* book for details on transferring string data.)

Register Operands

A register operand represents the contents of a register only. In these expressions, data is transferred directly from register-to-register, for example:

```
MOV   CH,DL
```

or from register-to-memory, for example:

```
SUB   MEM_BYTE,DL
```

Immediate Operands

An immediate operand contains a representation of the actual data being transferred; that is, the data is contained within the instruction itself. Data can be transferred immediate-to-register for example:

```
MOV   AX,8
```

or immediate-to-memory for example:

```
ADD   VAR[100*TYPE VAR],9
```

In the above examples, the source operands, 8 and 9, could have been enclosed within brackets ([ ]) without altering the meaning of the expression. However, bracketing of registers, for example, [BX], produces a different result than non-bracketing of the same expression (see "Coding Operands" later in this chapter for more information).

Storage Operand

A storage operand is either a code-relative label or a data-relative variable (defined by one of the DEFINE DATA pseudo-ops).

Data defined as a storage operand can be transferred either *direct*ly or *indirect*ly. Directly transferred data is identified by an address symbol (label or variable) within the instruction. Data is transferred memory-to-register, for example:

```
MOV   AX,TABLE
```

or from an indexed array, such as:

```
MOV   AL,TABLE[SI+BX][8]
```

or where an absolute value is defined as an EQUated symbol and used to index a table, such as:

```
VAR  EQU   8
     MOV   AL,TABLE[VAR]
```

Indirectly transferred data is identified by indexing (bracketing) of a *base* register (BX or BP), *index* register (SI or DI), *displacement* value (constant), or some combination of registers and/or constants. Bracketing the registers tells the assembler that a pointer (offset, not the data itself) is contained within the register. Data is transferred memory-to-register, through a base register (BX or BP), optionally with an 8-bit or 16-bit displacement, for example:

```
MOV AX,[BX]+8
```

or through an index register (SI or DI), optionally with an 8-bit or 16-bit displacement, for example:

```
MOV AX,4[SI]
```

or through the sum of one base register (BX or BP) and one index register (SI or DI) optionally with an 8-bit or 16-bit displacement, for example:

```
MOV AX,16[BX+SI]
```

Structure Fields

Structure fields are operands of the form *variable.field*. The *variable* entry is any variable used to identify and/or change a STRUC defined data block. A *field* entry is a variable defined by the DEFINE DATA pseudo-op within the targeted STRUC block. For example:

```
CTRLBLCK  STRUC
FIELDA    DW     ?
FIELDB    DW     ?
CTRLBLCK  ENDS
AREA      CTRLBLCK  <>
          MOV    AX,AREA.FIELDB
```

The STRUC/END pseudo-ops define the datablock, "CTRLBLCK." "FIELDB," defined by DW within "CTRLBLCK," is identified by the "AREA" variable and placed in the AX register. (See the STRUC pseudo-op in Chapter 2 of the *Macro Assembler Reference* book and the TRYCOM and SALEX examples on the Assembler Diskette for more details on structures.)

Coding Operands

Coding Storage Operands

The assembler's syntax requirements allows certain indexed storage operands to be coded in different ways. If the operand includes a register that contains the pointer (offset) of a data item (not the data itself) the assembler must be informed by enclosing the register within square brackets ([ ]). Within the brackets, however, any combination of base register, index register and displacement value is permitted. All of these operands, for example, produce the same result when used as the object of a "MOV" mnemonic:

```
MOV    AX,8[BX][SI]
MOV    AX,8[BX+SI]
MOV    AX,[BX][SI]+8
MOV    AX,[BX+SI]+8
MOV    AX,[BX+SI+8]
MOV    AX,[BX+8+SI]
MOV    AX,[BX][SI][8]
```

Also the displacement value, 8, can either be placed within the brackets, in front of the bracketed registers (using no operator), bracketed and placed before or after the bracketed registers, or appended to the bracketed registers with a plus (+) operator.

Coding Immediate Operands

Unlike bracketing of registers, bracketing of immediate values (constants) or symbols (variables or labels)–where no indexing of registers is denoted–produces the same result as non-bracketing of the same values. For example, the statements:

```
MOV    AX,[8]
MOV    AX,[SYM]
```

produce the same results as:

```
MOV    AX,8
MOV    AX,SYM
```

Bracketing an absolute number indicates the number is an index into an array of bytes, for example:

```
MOV   AL,BTABLE[8]
```

which is identical to:

```
MOV   AL,BTABLE+8
```

The same result is produced if an absolute value is defined as a variable and used as an index:

```
FOO   EQU   8
MOV   AL,BTABLE[FOO]
```

To achieve a memory reference by using an absolute offset, you can specify a segment register (the DS register is the default memory-accessing segment register) followed by a colon (:) and the bracketed value you want to use as an offset. For example:

```
   MOV   AX,DS:[8]
```

or:

```
FOO  EQU   8
     MOV   AX,DS:[FOO]
```

See "Construction Rules" in this chapter for more details on operand syntax.

Assembler syntax contradicts the syntax used by the DOS DEBUG UNASSEMBLE command. Unlike the assembler, DEBUG uses square brackets around an absolute number to indicate a memory reference instead of an immediate value. If you code a memory reference as:

```
   MOV   AX,DS:[8]
```

DEBUG unassembles this code as:

```
   MOV   AX,[8]
```

If you code:

```
   MOV   AX,[8]
```

DEBUG unassembles this code as:

```
   MOV   AX,8
```

Take special care when coding immediate-to-memory operands. For example, if you code:

```
   MOV  [BX],1
```

The assembler generates an "Operand must have size" error. Because the BX register is bracketed, it contains an offset location which the programmer must specify as 1 or 2 bytes. This task can be performed with the BYTE, WORD and PTR operators for example:

```
MOV   BYTE PTR [BX],1   ;MOVE BYTE IMMEDIATE
```

or

```
MOV   WORD PTR [BX],1   ;MOVE WORD IMMEDIATE
```

An immediate-to-register transfer can be accomplished without additional operators for example:

```
MOV   BL,1              ;MOVE BYTE IMMEDIATE
MOV   BX,[1]            ;MOVE WORD IMMEDIATE
```

In these examples, the assembler is able to make assumptions about what types of data are being transferred; however, it is the programmer's responsibility to ensure that the assembler's assumptions correspond with his intentions.

Operators

An operator is a symbol that is used at assembly time to qualify an operand expression. Six groups of operators are described here:

Attribute operators
Value-returning operators
Record-specific operators
Arithmetic operators
Relational operators
Logical operators.

Attribute Operators

Attribute operators can be used to override an operand's attributes, return the values of an operand's attributes, and isolate record fields. These operators are:

Pointer (PTR): The format of the pointer operator is:

type PTR *expression*

PTR can be used to override the type attribute (BYTE, WORD, DWORD, QWORD, or TBYTE) of a variable or the type attribute (NEAR or FAR) of a label. The *type* field is the new attribute. The *expression* field is the identifier whose type attribute is to be overridden. For example:

```
CALL WORD PTR [BX][SI]
```

Segment: The segment override operator overrides the segment attribute of a label, variable, or address expression. This operator takes three forms:

1. *seg-reg:addr-expr*

   Example:

   ```
   MOV AX,ES:[BX+SI]
   ```

2. *seg-name:addr-expr*

   Example:

   ```
   SYMBOL   EQU      4
   CODESEG  SEGMENT
            MOV      AX,CODESEG:SYMBOL
   ```

3. *group-name:addr-expr*

   Example:

   ```
   MOV AX,OFFSET DGROUP:VARIABLE
   ```

SHORT: The SHORT override operator changes the NEAR attribute of the targeted symbol. Only use SHORT when the symbol follows within 127 bytes of JMP. If the targeted symbol is coded before JMP, do not code SHORT; the assembler assumes SHORT if the symbol is within range. If SHORT is not coded, the assembler produces a NEAR JMP on Pass 1 and a SHORT JMP and NOP (to avoid phase error) on Pass 2. (See Chapter 5 "Macro Assembler" for details on Pass 1 and Pass 2.) Expressly coding the SHORT operator avoids the unneeded generation of the NOP. For example:

```
      JMP SHORT TAG
       o
       o
       o
  TAG:
```

THIS: The format of the THIS operator is:

THIS *type*

The THIS operator creates an operand whose attributes are:

- Segment attribute — equal to the segment in which it is defined.
- Offset attribute — equal to the current value of the location counter.
- Type attribute — equal to the value defined by the THIS operator.

For example, assume the symbol "TAG" is defined in the code segment as:

```
TAG EQU THIS BYTE
```

The type attribute BYTE overrides the assumed type of NEAR.

HIGH and LOW: These operators provide some 8080 capability. They are called the *byte isolation operators*. The HIGH and LOW operators accept a number or an an address expression as an argument. HIGH returns the high-order byte. LOW returns the low-order byte. For example:

```
MOV AH, HIGH WORD_VAR  ;GET BYTE WITH SIGN BIT
```

Value-Returning Operators

Value-returning operators are *passive*; they return values but do not override attributes:

SEG: The format of the SEG operator is:

SEG *variable* or *label*

The SEG operator returns the segment value of the variable or label. For example:

```
EXTRN  VARNAME:BYTE
MOV    AX,SEG  VARNAME
MOV    ES,AX
MOV    AX,ES:VARNAME
```

OFFSET: The format of the OFFSET operator is:

OFFSET *variable* or *label*

The OFFSET operator returns the offset value of the variable or label.

If you are using the GROUP pseudo-op, do not expect the OFFSET operator to yield the offset of a variable within the group. It returns the offset of the variable within its segment. If you need the offset of a variable within a group, use the GROUP name as the GROUP override operator instead, for example:

```
DGROUP GROUP   DATA, CSEG
DATA   SEGMENT
       o
VAR    DB      0
       o
       DW      VAR               ;GIVES OFFSET
                                 ; WITHIN SEGMENT
       DW      DGROUP:VAR ;GIVES OFFSET
                                   ; WITHIN GROUP
       DD      VAR                 ;INCORRECT
       DD      DGROUP:VAR ;CORRECT
DATA   ENDS
CSEG   SEGMENT
       ASSUME  CS:CSEG,DS:DGROUP
       MOV     BX, OFFSET VAR  ;LOADS SEG
                                ; OFFSET OF VAR
       MOV     BX, OFFSET DGROUP:VAR
                    ;LOADS GROUP OFFSET OF VAR
       o
       o
       o
CSEG   ENDS
```

The GROUP statement must precede all other uses of the group-name; that is, no forward references to group-names are permitted.

In most assemblers, the only attribute of a variable is its offset, so that a reference to a variable's name is a reference to its offset. Because this assembler defines five attributes for a variable, the OFFSET operator is required to isolate the offset value.

However, OFFSET is not required in a DW or DD pseudo-op. An implicit OFFSET is applied to variables in address expressions appearing in DW and DD pseudo-ops. For example:

```
TABLE_PREFIX_BYTE  DW  TABLE-1   ;OFFSET OF
                     ; BYTE PRECEDING TABLE
```

TYPE: The format of the TYPE operator is:

TYPE *variable* or *label*

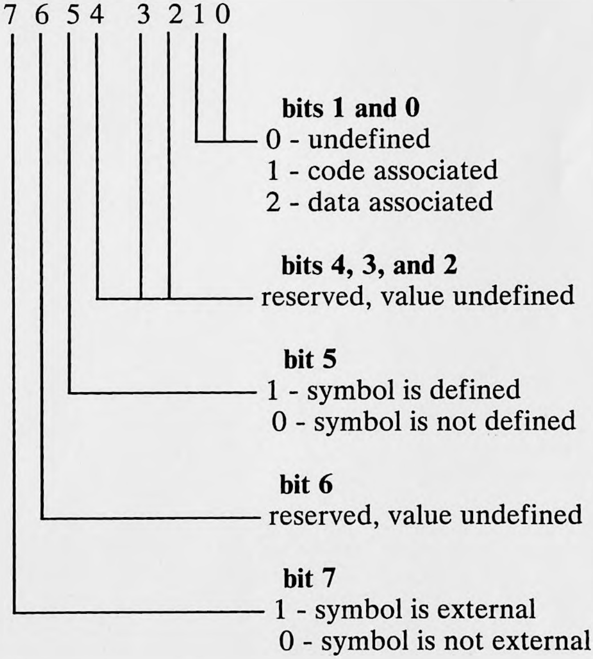
If the operand is a variable, the TYPE operator returns a value equal to the number of bytes of the operand:

- BYTE=1
- WORD=2
- DWORD=4
- QWORD=8
- TBYTE=10
- Structure name = the number of bytes declared by STRUC.

If the operand is a label, the TYPE operator returns the values:

> 65534 (which can be thought of as -2) for symbols whose attribute is FAR
>
> 65535 (which can be thought of as -1) for symbols whose attribute is NEAR

.TYPE: Do not confuse the .TYPE operator with the TYPE operator (no leading period). The .TYPE operator is often used within macros to determine the kind of parameter being passed. The value generated by this operator is defined as having bit flags in a byte value:


7 6 5 4 3 2 1 0
bits 1 and 0
0 - undefined
1 - code associated
2 - data associated
bits 4, 3, and 2
reserved, value undefined
bit 5
1 - symbol is defined
0 - symbol is not defined
bit 6
reserved, value undefined
bit 7
1 - symbol is external
0 - symbol is not external

SIZE: The format of the SIZE operator is:

SIZE *variable*

The SIZE operator is the product of LENGTH times TYPE. SIZE is only significant for single items being duplicated.

LENGTH: The format of the LENGTH operator is:

LENGTH *variable*

For a DUP entry, the LENGTH operator returns the number of units allocated for the variable. For all others, 1 is returned. For example:

```
EXP  DW  100  DUP(1)
```

The values returned are: LENGTH=100, SIZE=200, TYPE=2.

```
   MOV CX,LENGTH EXP   ;GET NUMBER OF BYTES
;                       IN ARRAY
```

Record-Specific Operators

Record-specific operators are used to extract or insert data into a record (see the RECORD pseudo-op in Chapter 2 of the *Macro Assembler Reference* book for details on records).

> **Note:** Because they are associated with the RECORD facility, record specific operators are not supported by the Small Assembler.

These are:

SHIFT COUNT: The format of this operator is:

recfield

Shift count is derived from the record field name to be isolated.

MASK: The format of this operator is:

MASK *recfield*

The MASK operator returns a bit mask defined as 1 for bit positions included in the field and 0 for bit positions not included in the field.

WIDTH: The format of this operator is:

WIDTH *recfield*

The WIDTH operator returns the width of a record or a record field as the number of bits in either the record or the field.

Arithmetic Operators

These operators are used between two terms in an expression and indicate the arithmetic operation to be carried out on these two terms.

Addition: The format of this operator is:

term + term

The two terms and the addition operator return the sum of the two terms. For example:

```
MOV  AX,[BX+DI]
MOV  AX,SYMBOL+2
```

Subtraction: The format of this operator is:

term - term

The two terms and the subtraction operator return the difference of the two terms. For example:

```
MOV  AX,100H - 2
```

Multiplication: The format of this operator is:

*term * term*

The two terms and the multiplication operator return the product of the two terms. For example:

```
LEA  BX,TABLE+(20*TYPE TABLE)
```

This example contains a compound expression. After evaluating the contents of the parentheses as the product of the two terms, that product becomes the second term in an addition operation, which is then evaluated. In this particular example, the parentheses are not needed because multiplication would be carried out before addition (see the "Precedence of Operators" section later in this chapter).

Modulo: The format of this operator is:

term MOD *term*

The two terms and the MOD operator return the remainder obtained by dividing the first term by the second term. For example:

```
MOV  AX,20 MOD 3  ;SAME AS, MOV  AX,2
```

Shift Left: The format of this operator is:

term SHL *term*

> **Note:** Do not confuse this operator with the machine instruction that uses the same mnemonic.

The two terms and the SHL operator return the value of the binary equivalent of the first term shifted left the number of bit positions indicated by the second term. Zero value bits are moved into vacated bit positions. For example:

```
MOV  AX,17 SHL 2
```

In binary notation, 17 (decimal) is 0001 0001. Shift this value left 2 positions and it becomes 0100 0100 which is 44H (or 68).

Shift Right: The format of this operator is:

term SHR *term*

> **Note:** Do not confuse this operator with the machine instruction that uses the same mnemonic.

The two terms and the SHR operator return the value of the binary equivalent of the first term shifted right the number of bit positions indicated by the second term. Zero value bits are moved into vacated bit positions. This is a logical shift (not an algebraic shift) because the sign bit is not moved into the vacated positions.

Relational Operators

These operators are used between two terms in an expression and return a value of TRUE (-1) or FALSE (0) depending on the relationship defined by the operator. There are six relational operators:

EQ (equal)
NE (not equal)
LT (less than)
LE (less than or equal)
GT (greater than)
GE (greater than or equal)

When relational operators are used, results are produced by comparing 17-bit signed numbers, where the 17th bit can contain only a sign bit. Thus the expression:

```
OFFFFH  GT  7FFFH
```

is true, while the expression:

```
-1  GT  7FFFH
```

is false.

Logical Operators

The AND, OR, and XOR operators are used between two terms. The specified operation is performed on each corresponding bit position to form one term. The NOT operator is used with only one term and is placed before the term.

AND: The format of this operator is:

term AND *term*

The results of this operation are:

BIT VALUE OF TERMS		**OPERATION**
A:	B:	A AND B
1	1	1
1	0	0
0	1	0
0	0	0

For example:

```
MOV   AX,FOO AND 7FFFH   ;CLEAR HIGH BIT FROM
;                         REG IMAGE OF FOO
```

OR: The format of this operator is:

term OR *term*

The results of this operation are:

BIT VALUE OF TERMS		**OPERATION**
A:	B:	A OR B
1	1	1
1	0	1
0	1	1
0	0	0

For example:

```
MOV   AX,FOO OR 8000H   ;SET HIGH BIT IN REG
;                        IMAGE OF FOO
```

XOR: The format of this operator is:

term XOR *term*

The results of this operation are:

BIT VALUE OF TERMS		OPERATION
A:	B:	A XOR B
1	1	0
1	0	1
0	1	1
0	0	0

For example:

```
MOV   AX,FOO XOR 1   ;TOGGLE REG IMAGE OF FOO
;                     FROM EVEN TO ODD, OR
;                     FROM ODD TO EVEN
```

NOT: The format of this operator is:

NOT *term*

The result of this operation is a bit by bit compliment, which is not the same as the negation of the term:

BIT VALUE OF OPERAND:	NOT OPERAND
0	1
1	0

For example:

```
MOV   AX,NOT FOO   ;TOGGLE ALL BITS IN
;                   REG IMAGE OF FOO
```

Do not confuse logical operators with the logical instruction mnemonics whose coding is identical. Operators perform at assembly time while instructions perform at execution time. Of course, logical instructions and operators can be used together, for example:

```
FLAGA   EQU   0400H
FLAGB   EQU   0080H
        o
        o
        o
        OR    FLAGWORD,FLAGA OR FLAGB   ;TURN
;             ON BOTH STATUS BITS IN FLAGWORD
```

Precedence of Operators

Expressions are evaluated with higher precedence operators first, then left to right for equal precedence operators. Parentheses can be used to change precedence. The precedence of operators in the order of highest to lowest priority is:

1. Entries inside parentheses (entry), square brackets [entry], the structures variable (*variable.field*), and then the operators LENGTH, SIZE, WIDTH, and MASK

2. The name: (segment override)

3. The PTR, OFFSET, SEG, TYPE, THIS, and segment override operators

4. HIGH and LOW operators

5. Multiply and divide: *, /, MOD, SHL, SHR

   For example:

   ```
   MOV AX,100 MOD 17 ;AX = 15 = 000FH,
   ;                  SINCE 100 = 5*17+15
   ```

   (MOD accepts only absolute-number operands)

   ```
   MOV AX,101B SHL (2*2) ;AX = 01010000B
   ;                     (SHIFT LEFT 4 BITS)
   ```

   Since operators are evaluated left-to-right:
   101B SHL (2*2) = 01010000B
   while
   101B SHL 2*2 = 00101000B

6. Add and subtract: plus, minus (both unary and binary)

7. Relational: EQ, NE, LT, LE, GT, GE

8. Logical NOT

9. Logical AND

10. Logical OR, XOR

11. SHORT

The Comment Field

The *comment* field entry begins with a semicolon and is generally used to describe the purpose of the statement. For example:

```
ASSUME    DS:DATASEG    ;POINT THE DATA SEGMENT
MOV       AX,DATASEG    ;REGISTER AT THE
MOV       DS,AX         ;SEGMENT IDENTIFIED
                        ;BY "DATASEG"
```

Construction Rules

Certain rules and conventions apply when constructing your program statements:

All entries:

- Use uppercase or lowercase letters interchangeably (except for the contents of quoted strings in operands where case is significant).
- Separate all entries with at least one blank or tab character; two or more consecutive blanks (or tabs) are also acceptable.
- Use the name-operation-operand-comment order for each statement (even if two or three fields constitute the statement).
- Do not write more than 132 characters per line (continuation of a statement on the following line is not allowed).
- Enter only one source code statement per line.

  Note: A statement does not have to begin on the first character position of a line.

Name field entries:

- Create name entries by using a combination of these characters:
  - Uppercase alphabetic characters:

    `A through Z`
  - Lowercase alphabetic characters:

    `a through z`
  - Numeric digits:

    `0 through 9`

    Note: Numeric digits are not allowed as the first character of a name.
  - Special characters:

    `? . @ _ $ %`

- Entries may be as long as the user desires; however, the assembler only recognizes the first 31 characters.

- Use a period only as a first character of a name. A period is also used as a separator between *variable* and *field* in a structure field used as an operand. (see the STRUC pseudo-op in Chapter 2 of the *Macro Assembler Reference* book and "Structure Fields" earlier in this chapter).

- You can begin an entry in any column, preceded only by any combination of blanks or tabs. (Usually, the name field begins in the first column.)

- For structured assembler source to be processed by the SALUT utility:
  - the name entry must begin in the first column.
  - only a name entry is allowed in the first column.

Operation field entries:

- Accurately spell all mnemonic entries to avoid unnecessary error messages.

- Avoid using mnemonics as name entries. For example:

  ```
  TITLE MACRO
  ```

  It is difficult to tell whether "MACRO" is redefining the TITLE pseudo-op or being defined as the text of a TITLE statement. If both Name and Operation look like a mnemonic, the second entry becomes the mnemonic and the first becomes the name, unless:

  - The name would be required to end with a colon, making it a NEAR label.

    or,

  - The name is a pseudo-op.

  For these exceptions, the first entry is the mnemonic, the second entry becomes its operand.

Operand field entries:

- For instruction mnemonics, separate the source and destination operands with a comma (,).
- For certain pseudo-ops, separate two or more operands with one or more blanks or tabs (see the individual pseudo-ops in Chapter 2 of the *Macro Assembler Reference* book for details).
- Elements joined by a plus (+) sign and placed within square brackets ([ ]) can be arranged in any order (see "Coding Operands" earlier in this chapter for details).
- Elements can be individually bracketed or placed entirely within one set of brackets and connected with a plus (+) operator.
- Immediate values (constants) can be bracketed or unbracketed without altering their meaning.
- If a displacement value is positioned in front of an indexed value, no plus (+) operator is necessary.
- If a displacement value is positioned after an indexed value, it must be preceded by a plus (+) operator.

Comment field entries:

- The semicolon identifies the start of the comment.
- A comment can either be on a line by itself or be the final field of any statement, instruction, or pseudo-op (see the "Statement Format" in the following section of this chapter for a description of fields).
- The assembler ignores the contents of a comment, therefore, it can contain any combination of characters.
- A totally blank line is a special kind of comment that provides vertical spacing in the listing, but is otherwise ignored.

> **Note:** COMMENT is also a pseudo-op (see the "COMMENT" pseudo-op in Chapter 2 "Pseudo Operations" of the *Macro Assembler Reference* book).

Statement Format

To avoid confusion, organize your statements into four fields (name-operation-operand-comment) so that each statement is accurately interpreted by the assembler. Aligning each entry with the beginning of its corresponding field also improves your program's readability and appearance. For example:

Formatted source code:

```
HERE:   MOV   CL,VAL1   ;MOVE VAL1 TO CL
        ADD   CL,VAL2   ;ADD VAL2 TO VAL1
        RET
```

Unformatted source code:

```
HERE: MOV CL,VAL1 ;MOVE VAL1 TO CL
ADD CL,VAL2 ;ADD VAL2 TO VAL1
RET
```

Clearly, the formatted source code is easier to read and understand.

> **Note:** The SALUT utility can format your source code into four fields. You must enter the "F" parameter in response to SALUT prompts (See the "F" parameter in Chapter 4 "SALUT" for more information).

Here are some additional suggestions for formatting statements:

- Only the name field should begin in column 1.
- Include comments with most statement entries to improve their understanding.
- Separate routines with blank lines to improve readability.

Register Code Conventions

Registers are known implicitly by the assembler and are reserved symbols. When a register is used in an instruction, it must be coded as a reserved symbol or as a user-defined symbol equated (using the EQU pseudo-op) to a reserved register name.

The 8088 register set is divided into four groups: general, pointer/index, segment, and instruction/flags.

General Registers

The general register set is composed of four 16-bit registers that can also be used as eight 8-bit registers. These registers are:

16-Bit Registers	**8-Bit Registers**	
	High	**Low**
AX (Accumulator)	AH	AL
BX (Base)	BH	BL
CX (Count)	CH	CL
DX (Data)	DH	DL

The "X" identifier of each 16-bit register name indicates that the register is used in its full 16-bit state. The "H" identifier of each 8-bit register indicates that the high end only (upper 8 bits) of the 16-bit register is being used. Likewise, the "L" indicates that the register is the lower 8 bits of a 16-bit register.

Pointer (Offset) Registers

Although pointer registers can be used to perform most 16-bit arithmetic and logical operations, these registers are usually used as offset pointers to data in memory (see "Storage Operands" in this chapter for more details). These registers are:

Base Pointers

BX	Data segment base pointer
BP	Stack segment base pointer

Index Pointers

SI	Source index pointer
DI	Destination index pointer

Stack Pointer

SP	Pointer to newest active element in the Stack Segment

Memory accessing instructions can point to a memory location by using a Base Pointer, or an Index Pointer, or by using both one Base Pointer and one Index Pointer (using the sum of their contents to point to a memory location). No more than one Base or one Index pointer can be used in any one memory referencing expression.

Instruction Pointer

IP	Pointer to next instruction to be executed in the Code Segment

The Instruction Pointer is not directly addressable, but is continually updated to point to the next instruction after the one currently executing. That value may be changed by the execution of an instruction, such as: JMP, CALL, RET.

Segment Registers

Your IBM Personal Computer's memory is addressable only through segments. A segment is a contiguous block of memory up to 64K long that begins at a paragraph boundary. (See Chapter 3 "Memory and Segmentation" for details on paragraphs). To identify the beginning of each segment, segment registers hold the beginning paragraph numbers of segments. In .EXE structured files there are four *current* segments. The paragraph number of each segment is called the *segment base value* (or *segment identifier*) and is contained in its corresponding segment register as:

- ES register—can define an auxiliary data segment
- CS register—always defines the current code segment
- SS register—always defines the current stack segment
- DS register—usually defines the current data segment

The ASSUME pseudo-op must be used to tell the assembler which segment register to use when addressing certain segments. For example:

```
ASSUME  CS:CODESEG
```

This example causes the assembler to establish the CS segment register as the base address (segment identifier) of all statements defined within the segment "CODESEG."

Flags Register

The flags register (flag register word) is a 16-bit register that contains six 1-bit status flags and three 1-bit processor-control flags. The flags are ordered within the register as:

```
x  x  x  x  OF DF IF TF SF ZF x  AF x  PF x  CF
15                    8 7                     0
```

x = Reserved

CF = Carry Flag
PF = Parity Flag
AF = Auxiliary Carry Flag
ZF = Zero Flag
SF = Sign Flag

TF = Trace Flag
IF = Interrupt Flag
DF = Direction Flag
OF = Overflow Flag

Status flags

After a given logical or arithmetic instruction has executed, the status flags are set to 1's or cleared to 0's to reflect the result of the operation. The status flags are defined as:

CF The carry flag is set to 1 if the operation results in a carry out of (from addition) or a borrow into (from subtraction) the high-order bit of the result; if not, this bit is set to 0.

PF The parity flag is set to 1 for even parity; it is set to 0 for odd parity.

AF The auxiliary carry flag is set to 1 if the operation results in a carry out of (from addition) or borrow into (from subtraction) the low-order four bits of the result; if not, this bit is set to 0.

ZF The zero flag is set to 1 if the result of the operation is zero; if not, this bit is set to 0.

SF The sign flag is set to 1 if the high-order bit of the result is set to 1; if not, this bit is set to 0.

OF The overflow flag is set to 1 if the signed operation results in an overflow; if not, it is set to 0.

Processor control flags

The processor-control flags are used to change processor operations. The control flags are defined as:

TF The trace flag puts the system in a single-step mode for debugging purposes. An internal interrupt is generated after each instruction so that you can inspect your program instruction-by- instruction.

IF The interrupt-enable flag allows recognition of external (maskable) interrupt requests. The interrupts are disabled when the IF bit is cleared to 0. The instruction, STI, enables interrupts, while CLI disables interrupts. The INT instruction also disables interrupts by clearing the IF bit.

DF When the direction flag is set to 1 (by the STD instruction), SI and DI are decremented; if set to 0 (by the CLD instruction), SI and DI are incremented. This bit is used for the MOVS, MOVSB, MOVSW, CMPS, CMPSB, and CMPSW instructions.

Implicit Use of Registers

Some instructions cause certain registers to be used implicitly. Whenever the instruction is executed, a given register is employed to perform a function related to the execution of the instruction. When a register is used implicitly, its contents are manipulated according to the specifications of the instruction. It is important, therefore, that you be aware of the instructions that use registers implicitly so the contents of those registers can be saved, if necessary, prior to the use of the instruction. Here is a list of some commonly used instructions and the registers they employ:

REGISTER	OPERATIONS
AX	Word Multiply, Word Divide, Word I/O
AL	Byte Multiply, Byte Divide, Byte I/O, Translate, Decimal Arithmetic
AH	Byte Multiply, Byte Divide
BX	Translate
CX	String Operations, Loops
CL	Variable Shift and Rotate
DX	Word Multiply, Word Divide, Indirect I/O
SP	Stack Operations
SI	String Operations
DI	String Operations

When a location is referenced indirectly through a base (BX or BP) or index (SI or DI) register (and no segment override prefix is specified) the assembler has to assume the segment in which the location resides. If the SI, DI, or BX register contains the offset of a location, the assembler assumes that the offset is in a segment based on the DS register. The DS register can be overridden with a segment override prefix (see examples below). One exception is when the DI register is the destination pointer for a string operation (such as, MOVS). When DI is used in this manner, the assumed segment register for DI is the ES register. In this case, the use of ES cannot be altered by a segment register override. If the BP register is used to reference a location, the implied segment register is SS. For this reason, BP is often used to contain an offset into the stack segment. The SS register can be overridden with a segment override prefix.

Examples:

```
MOV  AX,[BX]       ;DATA COMES FROM DATA
                   ;SEGMENT BASED ON DS
MOV  AX,[SI]       ;ALSO FROM DS
MOV  AX,[DI]       ;ALSO FROM DS
MOV  AX,[BP]       ;THIS DATA COMES FROM STACK
                   ;SEGMENT BASED ON SS
MOV  AX,CS:[BP]    ;EXPLICIT OVERRIDE IGNORES
                   ;SS DEFAULT AND USES
                   ;CS INSTEAD
MOV  AX,[BP+SI]    ;WHERE DEFAULTS MAY
                   ;APPEAR TO CONFLICT,
                   ;THE DEFAULT SEG-REG
                   ;ASSUMED BY THE BASE
                   ;OFFSET (BP)
                   ;IS USED.  IN THIS CASE,
                   ;IT IS THE SS REG.
MOVS BYTE PTR [DI],[SI]
                   ;THIS STRING OPERATION
                   ;MOVES DATA FROM THE
                   ;SI OFFSET INTO THE
                   ;DATA SEGMENT (DS) AND
                   ;FROM THE DI OFFSET INTO
                   ;THE EXTRA SEGMENT (ES).
MOVS BYTE PTR [DI],CS:[SI]
                   ;OF THESE TWO
                   ;SEGMENTS, ONLY THE SOURCE
                   ;CAN BE OVERRIDDEN.
```

The individual instructions in Chapter 3 of the *Macro Assembler Reference* book give details on the registers used implicitly (if any) for each instruction.

Chapter 3. Memory and Segmentation

Contents

Addressing Memory **3-3**

Program Structure **3-6**
  .EXE File Structure 3-7
    Stack Segment 3-7
    Data Segment 3-9
    Code Segment 3-11
  .COM File Structure 3-17

.EXE and .COM File Comparison **3-21**

Addressing Memory

Each memory location allocated to your application is represented by two registers—a segment register and an offset register. These registers are:

- Segment registers — CS,DS,ES,SS
- Offset registers — DI,SI,BP,IP,SP,BX

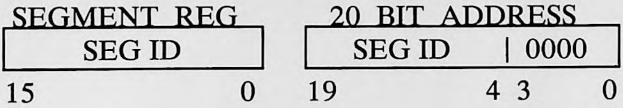
Because a register is 16-bits long, it is capable of representing unsigned integers between 0-65535 (64K) (2^{16} = 64K). Clearly, a segment register, by itself, does not contain enough bits to represent memory locations greater than 64K—hence, a second register, an offset register, is used in combination with the segment register to represent a memory location within the memory space.

To see how your two-register addressing scheme works, let us first review how memory is divided.

Memory is divided into 16-byte sections called *paragraphs*. Beginning at zero, each paragraph increments by 16 (10H) bytes until the memory capacity of one megabyte (1M) is reached. For example:

PARA-GRAPH	**DECIMAL VALUE**	**BEGIN ADDRESS**	**MEMORY**
0	0	00000H	16 BYTES
1	16	00010H	
2	32	00020H	
•	•	•	•
•	•	•	•
•	•	•	•
64k-1	1MB-16	FFFF0H	

Notice that the value of the beginning address of each paragraph contains 5 hex characters (followed by an "H" for hexadecimal), the last of which is always zero. This zero is a representation of 4 bits (one half-byte) that do not actually reside in the register but are used to show that the final representation of a given location's address is 20 bits long instead of 16.



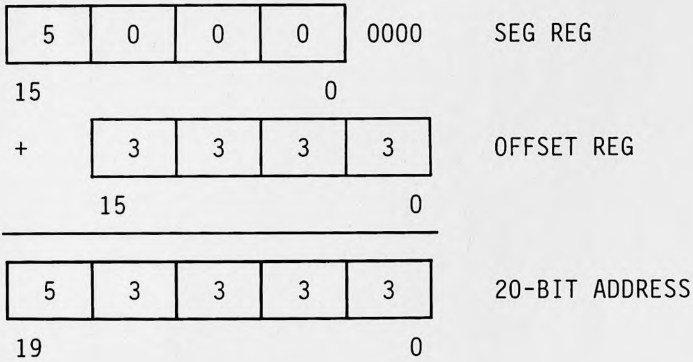
Using a segment register with an assumed, appended zero half-byte allows the representation of one paragraph boundary each time the hexadecimal value is increased by 1. Thus, we now have the means of accessing each paragraph in memory beginning at zero (00000H) and ranging up to 1 megabyte (FFFF0H).

Now that we have a 20-bit scheme to access any paragraph in memory, we can define a section of memory in which each location can be accessed. This defined section of memory is called a *segment.* A segment is a contiguous block of memory up to 64K long that begins at a paragraph boundary. To represent the beginning of a segment, we use a 16-bit segment register (with an appended, assumed zero half-byte). The paragraph value that is placed within a 16-bit segment register is called a *segment identifier* (*seg ID*) (see previous illustration). This value is also known as the *segment base value*

> **Note:** All Macro Assembler programs must have at least one defined segment (see the "Program Structure" section later in this chapter for details).

We can now use a second register (an offset register) to identify a specific location within a segment. The value within an offset register is called an *offset.* An offset represents the number of bytes from the segment base value to a location within a segment. Because a segment is no larger than 64K, a 16-bit offset register can represent any location within a segment.

These two values—the seg ID and the offset—are added together to produce a 20-bit *physical address.* in this manner:


5 0 0 0 0000
SEG REG
15
0
+
3 3 3 3
OFFSET REG
15
0
5 3 3 3 3
20-BIT ADDRESS
19
0

Program Structure

Executable files can be broadly grouped into two categories:

- Multi-segment (.EXE) files
- Single-segment (.COM) files

  Note: Programs may be referred to as "files" in this section (see Chapter 1 "The Application Development Process" for details on file identification).

.EXE files are organized into separate, individual segments. .COM files contain only one segment, which includes all data and instruction code.

There are advantages to both types of files. For example, .COM files avoid the overhead generated by separate segment identification, while .EXE files can accommodate much larger applications. Determine which file type is best suited for your particular application before you begin writing source code.

> **Note:** See the list at the end of this section and the enclosed examples, SKELEXE.ASM, SKELEXE2.ASM, and SKELCOM.ASM (on the Macro Assembler Diskette), for more information on the file differences.

.EXE File Structure

An .EXE file is generally composed of three segments:

- STACK segment
- DATA segment
- CODE segment

A fourth segment can also be defined and addressed with the ES register. Some applications and instructions, for example, MOVS, require the use of this segment (see "Register Code Conventions" in Chapter 2 "Assembly Language Format").

Stack Segment

The stack segment must be defined in an .EXE executable program file. The stack segment may appear in any order relative to the other segments. The application uses the stack segment to store any data that must be saved and then recalled later in the program.

The stack segment is made up of three statements:

- Beginning statement
- Define size statement
- Ending statement

The beginning statement defines the segment name, beginning boundary, type and group class of the stack segment. For a stack segment, the combine-type must be "STACK" (see "SEGMENT" in Chapter 2 "Pseudo-Operations" in the *Macro Assembler Reference* book for details.) For example:

```
STACKSEG   SEGMENT   PARA STACK 'STACK'
```

The define size statement defines the size of the stack segment by using the "DB" (Define Byte) or "DW" (Define Word) operation mnemonics. Here are three examples of define size statements:

```
DB   32  DUP ("STACK...")
DB   256 DUP (0)
DW   128 DUP (0)
```

All the stack operations transfer 16-bit *words*. Accordingly, the SP (Stack Pointer) register is decremented by two when an operand is "pushed" onto the stack and incremented by two when an operand is "popped" off the stack (see "PUSH" and "POP" in Chapter 3 "Instruction Mnemonics" in the *Macro Assembler Reference* book). When defining the stack size with the DB mnemonic, therefore, always use an *even* number of bytes.

The ending statement signals the end of the stack segment by repeating the *segname* and adding the 'ENDS' (end segment) mnemonic. For example:

```
STACKSEG    ENDS
```

Data Segment

Like the stack segment, the data segment typically consists of a beginning SEGMENT statement (except the combine-type is not "STACK" but usually "PUBLIC") and an ending ENDS statement. The body of the data segment defines the variables referenced in the code segment. For example:

```
SUM   DB  ?  ;RESERVE 1 BYTE FOR VARIABLE 'SUM'
VAL1  DB  1      ;VAL1=1 (DECIMAL)
VAL2  DB  0FH    ;VAL2=15 (DECIMAL)
MSG   DB  'MESSAGE  K. W. C.',13,10,'$'
```

Because the code segment refers to variables defined in the data segment, it is important to place the data segment before the code segment. By doing so, you decrease the possibility of generating assembler errors and increase the speed and efficiency of program execution.

Unless otherwise specified, segments are re-ordered after the assembly, alphabetically. A segment defined as "CODESEG", therefore, is placed prior to a segment defined as "DATASEG" unless the assembler is instructed to override its default. Specifying the /S parameter at assembly time causes segments to be ordered in the same sequence in which they are listed in the source file.

When using the /S parameter, there may be circumstances requiring segments to be assembled in an order other than their alphabetic sequence within a source file. Assume, for example, you are assembling a module in which the data segment precedes the code segment. If you want to maintain the segment order of this file but enable the code segment to be placed before the data segment, you can place a dummy segment with the same name as the current code segment before the data segment. The dummy segment needs only two statements, for example:

```
CODESEG   SEGMENT   PARA PUBLIC 'CODE'
CODESEG   ENDS
```

At assembly time, any segments with names that are identical to previously defined segments are moved next to the originally defined segment. (see the "Parameters" section of Chapter 5 "Macro Assembler" for details on the /A and /S parameters).

Code Segment

The code segment can also be defined using a combination of the align-type, combine-type and class attributes. The attributes that you assign to the beginning statement are dependent on the purpose of your program.

The code segment contains the set of executable instructions, beginning with some label identified as the entry point. All labels (NEAR or FAR) are defined within the code segment, and require that an ASSUME statement indicates the use of the CS register for its base.

Once all segments comprising a program have been defined, a means of establishing addressability of each of the segment's statements can be established. To understand the purpose of the statements that carry out this task, however, let us first review the DOS procedure that sets up the segment registers for program execution.

To execute an .EXE file, DOS sets up a block of memory called the Program Segment Prefix (PSP) that interfaces (connects) DOS with the program. The beginning word of the PSP contains the instruction (INT 20H) that the application uses to pass control out of the program. (See the "Appendixes" section of the DOS book for more details on the PSP.) DOS then sets into memory the code segment, data segment and stack segment of the program (not necessarily in that order). Finally, DOS places the seg IDs of the PSP, code segment and stack segment into the segment registers:

SEGMENT ID of:		**SEGMENT REGISTER**
PSP	→	DS and ES
CODE SEGMENT	→	CS
STACK SEGMENT	→	SS

For details on segment registers see "Register Code Conventions" in Chapter 2 "Assembly Language Format" and "SEGMENT" in Chapter 2 "Pseudo Operations" in the *Macro Assembler Reference* book.

Because DOS puts the seg IDs of the code segment and stack segment into the CS and SS registers, the assembler can be told what these registers contain by using the ASSUME pseudo-op. The ASSUME pseudo-op is usually placed immediately after the code segment statement. For example:

```
ASSUME    CS:CODE_SEG
ASSUME    SS:STACK_SEG
```

The individual code segment statements can now be addressed using the CS register and the IP register (Instruction Pointer) as the offset of each individual instruction.

The procedure (PROC) statement follows the ASSUME statement. This statement defines a block of source code that is terminated with an ENDP statement. The PROC mnemonic always defines a NEAR or FAR *type* attribute (see "PROC" in Chapter 2 "Pseudo Operations" in the *Macro Assembler Reference* book). As the first procedure containing the entry point from DOS, this PROC must be defined as FAR. The FAR attribute tells the assembler this procedure's segment has been entered from another segment.

Format:

procname PROC *type*

The first step of execution is to save the vector pointing to the beginning word of the PSP so that control can return to DOS when execution has completed. These instructions carry out this task:

```
PUSH      DS        ;SAVE SEG ID OF PSP
                    ;ONTO STACK
XOR       AX,AX     ;LOAD BEGINNING OFFSET OF
                    ;ZERO INTO A GENERAL REG
PUSH      AX        ;SAVE OFFSET OF ZERO
                    ;ON STACK
```

When the main procedure's RET (Return) instruction is processed, its FAR attribute causes the seg ID and offset (stored on the stack in this example) to POP. Control then returns to DOS when the INT 20H instruction (at offset zero in the PSP) is executed.

DOS initially points the DS register at the PSP and not at the data segment. Some applications can require the DS register to be retained as a PSP pointer. In such cases, the ES (Extra Segment) register can be used to address the data segment. Let us assume, however, that the DS register is used to point to the data segment. These statements carry out this task:

```
MOV     AX,data-segname  ;LOAD SEG ID INTO
                         ;A GENERAL REGISTER
MOV     DS,AX            ;PUT SEG ID INTO
                         ;THE DATA SEGMENT REG
ASSUME  DS:data-segname  ;TELL ASSEMBLER THAT
                         ;DS HAS SEG ID
                         ;OF DATA SEG
```

The MOV instructions place the seg ID of the data segment into the DS register. The ASSUME pseudo-op completes the process by telling the assembler it has addressability to items in the data segment. You have now completed the initialization of your .EXE file. Complete the body of your code segment by including the statements your particular application requires.

To end the code segment, include these required instructions and pseudo-ops:

- Return control to DOS with the RET instruction before ending the main procedure (or outermost procedure).
- End all blocks of code defined with PROC pseudo-ops with ENDP (End Procedure).
- Ensure that each called procedure issues a RET (return) to its caller.

Example:

```
MAIN   PROC   FAR
       o
       o
       CALL   SUBR
       o
       o
       RET           ;RETURN TO DOS
MAIN   ENDP
SUBR   PROC   NEAR
       o
       o
       RET           ;RETURN TO MAIN PROC
SUBR   ENDP
```

- End the code segment with ENDS (End Segment).

- End the program with an END pseudo-op.

  Note: The END pseudo-op can also define the entry point of the program if this module is receiving control from DOS. If the application is composed of two or more modules that must be linked, the secondary modules must not have an operand in their END statement(s). (See "END" in Chapter 2 "Pseudo Operations" in the *Macro Assembler Reference* book).

Here is how a code segment might end:

```
            o
            o
SEGNAME     ENDS            ;END THE SEGMENT
                            ;END PROGRAM, ENTER
            END     MAIN    ;AT "MAIN"
```

Now let us see how all of the sections combine to form a completed .EXE structured file:

```
STACKSEG  SEGMENT  PARA STACK 'STACK' ;COMMENTS
          DB    32 DUP ("STACK...")         "
STACKSEG  ENDS                              "
;
DATASEG   SEGMENT  PARA PUBLIC 'DATA'       "
          o
    (define variables)
          o
DATASEG   ENDS                              "
;
CODESEG   SEGMENT  PARA PUBLIC 'CODE'       "
          ASSUME   CS:CODESEG,SS:STACKSEG   "
MAIN      PROC     FAR                      "
;
          PUSH     DS                       "
          XOR      AX,AX                    "
          PUSH     AX                       "
;
          MOV      AX,DATASEG               "
          MOV      DS,AX                    "
          ASSUME   DS:DATASEG               "
;
          o
    (instruction code body)
          o
          RET                               "
MAIN      ENDP                              "
;
SUBR      PROC        NEAR                  "
          o
    (subroutine body)
          o
          RET                               "
SUBR      ENDP                              "
;
CODESEG   ENDS                              "
          END         MAIN                  "
```

This completes our discussion of .EXE files. For more details and examples of .EXE file structure, see the SALUT sample session in Appendix D and see the source file examples "SKELEXE.ASM and SKELEXE2.ASM" on the Macro Assembler Diskette.

.COM File Structure

.COM files, unlike .EXE files, contain only one defined segment. This "common" segment contains all program data—even the Program Segment Prefix (PSP) and the Stack. (See the "Appendixes" section of the DOS book for more details on the PSP.) Define this segment with a beginning SEGMENT statement and use the ASSUME pseudo-op to point all four segment registers at the code segment. For example:

```
CODESEG    SEGMENT  PARA PUBLIC 'CODE'
           ASSUME   CS:CODESEG,DS:CODESEG
           ASSUME   ES:CODESEG,SS:CODESEG
```

DOS builds the PSP but includes it in the segment as the first 256 bytes. Because the PSP is included in the segment, the entry point into the program must be defined at 100H (the end of the 256 byte-block). The ORG and END pseudo-ops and a program entry label (with a NEAR type attribute indicated with a colon (:)) combine to carry out this task. For example:

```
ORG     100H
BEGIN:
o
o
END     BEGIN
```

"ORG 100H" tells the assembler to skip the PSP. "END BEGIN" tells the assembler that "BEGIN:" (type NEAR) is the program entry point.

After defining the entry point into the program, you must connect this entry point to the starting statement of the defined segment (or the point at which you want the program to begin executing). If any data has been written between the entry point and the starting statement, this data must be skipped over to reach the starting statement. The JMP (Jump) instruction can do this task. For example:

```
BEGIN:  JMP   START   ;SKIP DATA AREA
```

The "JMP" instruction extends the entry point to "START", where program execution begins.

To define the stack, DOS puts the seg ID (common to all segments) into the SS register and sets the SP (stack pointer) register to the upper end of this common segment. DOS then pushes a zero (0) word onto the stack.

> **Note:** If adequate memory is available, the SP register points to the end of a 64K segment (the maximum size of a .COM file). If 64K memory is not available, DOS points the SP register at the end of memory.

Although data, mnemonics and stack operands reside in one common segment, you should define data items before they are referenced by source code mnemonics (after the "JMP START" statement and before the "START PROC" statement in the examples). Defining data items before they are referenced avoids unnecessary forward referencing that can reduce your application's speed and efficiency.

Once the data is defined, the primary procedure (PROC) can de defined using the PROC pseudo-op. For example:

```
START        PROC      NEAR
```

> **Note:** All procedures in a .COM file must be defined as NEAR (intrasegment) since all data and code are in one common segment.

You have now completed the initialization of your .COM file. Complete the body of your code segment by including the statements your particular application requires.

To end your .COM program, follow the rules for ending an .EXE program, but recall these distinctions:

- The END statement must identify the entry point label at offset 100H.
- All PROCs must be labeled NEAR.
- The RET at the end of the START procedure is a NEAR return (defined within a NEAR procedure). It pops the zero word from the stack (pushed there by DOS) to return to the INT 20H instruction at offset zero in the PSP.
- If your application consists of two or more modules, each module must have the same segment label.
- The END statement of all secondary modules must not identify the entry point label.
- There are no stack or data segments.

  Note: LINK gives an error message warning about the missing stack. For .COM files, expect this warning and ignore it.

- The .EXE file produced by the LINK utility must be converted to a .COM file by the DOS utility EXE2BIN, after which you should erase the .EXE file. The .EXE file is itself not executable.

Here is how a completed .COM program might look:

```
CODESEG        SEGMENT PARA PUBLIC 'CODE'
               ASSUME  CS:CODESEG,DS:CODESEG
               ASSUME  ES:CODESEG,SS:CODESEG
               o
               o
               ORG     100H
BEGIN:         JMP     START
               o
       (define data)
               o
START          PROC    NEAR
               o
    (instruction code body)
               o
               RET
START          ENDP
CODESEG        ENDS
               END     BEGIN
```

This completes our discussion of .COM files. For more details and examples of .COM file structure, see the TRYCOM sample session in Appendix D and the source file example "SKELCOM.ASM" on the Macro Assembler Diskette.

.EXE and .COM File Comparison

Here is a list to contrast the features of .EXE and .COM files.

	.EXE File	**.COM File**
Maximum Size:	depends on amount of available memory	64K
Entry Point:	anywhere	100H (end of PSP)
END defines entry point:	required	required
Separate stack segment:	required	not allowed
Multiple segments:	allowed	not allowed
Entry label has attribute of:	FAR	NEAR
Contents of stack at entry:	nothing	zero word
Return to DOS via:	RET (FAR)	RET (NEAR), or INT 20H, or INT 27H, or INT 21H (function 0, 31H)
Subroutine's type:	NEAR or FAR (depending on what segment CALL is in)	NEAR (all modules linked must define their code and data in the same named segment)

	.EXE File	.COM File
.EXE file produced by Linker is:	executable	non-executable, must be converted to .COM via EXE2BIN, then deleted
PSP is:	separate from load module's segment and is pointed to by ES and DS at entry	first 256 bytes of the one common segment, pointed to by all four segment registers

As set by DOS loader, initial value of:		
Segment Register	**.EXE File**	**.COM File**
Code Segment CS	SEG ID of segment whose entry is defined by END	SEG ID of the one common segment
Stack Segment SS	SEG ID of segment defined by Stack combine-type	SEG ID of the one common segment
Data Segment DS	SEG ID of the PSP	SEG ID of the one common segment
Extra Segment ES	Same as DS	SEG ID of the one common segment

Chapter 4. Structured Assembly Language Utilities (SALUT)

Contents

Introduction **4-3**

Structure Statements **4-4**
    Structure Statement Conditions 4-5
  Structure Statement Syntax 4-6
  Logic Flow Structures 4-9
    Logic Flow Structure Examples 4-10

How To Use SALUT **4-20**
    Structure Statement Usage 4-20
    Sample Development Process 4-20
  Starting SALUT 4-32
    Non-Interactive Mode 4-33
    Prompt Mode 4-35
  Error Messages 4-38

Introduction

The Structured Assembly Language Utilities (SALUT) is an aid in designing and printing assembler programs. SALUT has three optional functions:

- Convert Structure statements into assembler source

- Format and indent assembler source

- Replace tab characters with blanks

SALUT enables you to use easily recognizable, high level Structure statements to perform critical control procedures such as, comparisons, loops, and searches. Combined with the format and indent functions, SALUT Structure statements help you develop structured, understandable, easy-to-read assembler programs.

SALUT's primary function is to convert the 13 Structure statements (defined below in "Structure Statements") into assembler source instructions. The resultant file (identified by the filename extension .ASM) contains only assembler source code, that can be processed by the assembler.

The two secondary functions of SALUT help the user create a source file that is easy to read and print. If the *Format* option is selected, SALUT reads the source file and builds a new source file that is formatted into the four statement fields (see Chapter 2 "Assembly Language Format") and properly indented to identify instructions within each Structure. The *Replace Tabs* option reads your source file and replaces all the tab characters with blanks (spaces). Removing tabs results in larger output files, but eliminates problems when printing the files. Depending on which editor is used to write your source file, this option may or may not be needed. (Some editors insert tab characters for blanks to conserve space.)

Structure Statements

$IF	condition[,AND/OR][,LONG]
$ELSE	[LONG]
$ENDIF	
$DO	[COMPLEX[,LONG]]
$ENDDO	[condition/LOOPxx[,AND/OR]][,LONG]
$SEARCH	[COMPLEX[,LONG]]
$EXITIF	condition[,AND/OR][,LONG][,NUL]
$ORELSE	[LONG] *not used with $EXITIF NUL
$ENDLOOP	[condition/LOOPxx[,AND/OR]][,LONG]
$ENDSRCH	
$LEAVE	condition[,AND/OR][,LONG] *after $DO, $SEARCH, $ORELSE or $LEAVE
$STRTDO	*after $DO or $LEAVE
$STRTSRCH	*after $SEARCH, $ORELSE or $LEAVE

Where square brackets, [ ], represent options.

Where "condition" is one of: A, AE, B, BE, C, E, G, GE, L, LE, NA, NAE, NB, NBE, NC, NE, NG, NGE, NL, NLE, NO, NP, NS, NZ, O, P, PE, PO, S, Z, and (in certain cases described later) CXZ or NCXZ.

And where "LOOPxx" is one of: LOOP, LOOPE, LOOPNE, LOOPNZ, or LOOPZ.

Structure Statement Conditions

Condition	Description	Flags Tested
A	above	CF=0 and ZF=0
AE	above or equal	CF=0
B	below	CF=1
BE	below or equal	CF=1 or ZF=1
C	carry	CF=1
CXZ	CX register is zero	(CF or ZF)=0
E	equal	ZF=1
G	greater	ZF=0 and SF=OF
GE	greater or equal	SF=OF
L	less	(SF xor OF)=1
LE	less or equal	((SF xor OF) or ZF)=1
NA	not above	CF=1 or ZF=1
NAE	not above nor equal	CF=1
NB	not below	CF=0
NBE	not below nor equal	CF=0 and ZF=0
NC	not carry	CF=0
NE	not equal	ZF=0
NG	not greater	((SF xor OF) or ZF)=1
NGE	not greater nor equal	(SF xor OF)=1
NL	not less	SF=OF
NLE	not less nor equal	ZF=0 and SF=OF
NO	not overflow	OF=0
NP	not parity	PF=0
NS	not sign	SF=0
NZ	not zero	ZF=0
O	overflow	OF=1
P	parity	PF=1
PE	parity even	PF=1
PO	parity odd	PF=0
S	sign	SF=1
Z	zero	ZF=1

"Condition" indicates the type of conditional jump that is produced. The user written instructions (for example, ADD, TEST, OR, CMP) set the condition flags in the Flag register. The Structure statement generates code that tests the resulting settings of the condition flags in the Flag register.

The AND/OR allows the user to code multiple *Structure statements* that are evaluated using logical AND or OR to join them together. They are evaluated in the order which they occur. (See SALEX1.SAL for an example.)

The Structure statements, that allow the LONG option, default to short jumps when generating the Structures. A LONG parameter is required if the target label is not within -128 to +127 bytes of this instruction.

The "NUL" in $EXITIF means there is no "success" process and no $ORELSE. If "NUL" is used or any continuation $EXITIF, it must be on all $EXITIFs in this continuation series.

Any symbol in column 1 is assumed to be a variable name or label. The first symbol (other than semicolon) in a line not beginning in column 1 is assumed to be an instruction mnemonic.

Do not use the COMMENT pseudo-op with SALUT. Use the *semicolon* in column 1 of all block comment statements.

Optionally, the user may code a label in column 1 of any Structure statement. The presence of such a label does not affect the generated code, since no references are generated to that label. Such a label must be ended by a colon. In the conversion of the Structure statement to a comment, the first character of the user-specified label is replaced with a semicolon. The user-specified label is placed on a line by itself, just before the generated code.

> **Note:** When SALUT processes the Structure statements, the original Structure statement is converted to a comment line in the assembler source for later reference.

Because the NCXZ/CXZ condition causes the generation of a JCXZ instruction that has no reverse condition (there is no JNCXZ instruction), care must be taken in the use of this conditional. Use it only in those cases where the JCXZ can be properly generated. If an attempt is made to use the incorrect one of this pair of conditionals in a particular situation, SALUT generates an "Invalid Conditional" error message. For example, $LEAVE always generates the true condition. In this case, the condition can only be "CXZ". $LEAVE just puts the "J" in front to generate the desired "JCXZ" instruction. On the other hand, $IF always generates the opposite of the specified condition. SALUT selects the opposite of the "NCXZ", generating the desired "JCXZ" instruction.

For example:

- Assume CX has the count. A loop is to be carried out using that count. However, if the count is already zero, do not do any of the loop.

```
$IF             ;IF THE COUNT IS NON-ZERO
  $DO
    o           ;PERFORM BODY OF THE LOOP
    o
    o
  $ENDDO  LOOP  ;DECREMENT COUNT,
;                REPEAT ON NON-ZERO
$ENDIF
```

If the $IF/$ENDIF were left off, and the count in CX was zero, the loop would be carried out 64K times, which is not what was intended.

- Another example works, but has undesirable execution overhead:

  ```
  $DO
  $LEAVE  CXZ
    o
    o
    o
  $ENDDO LOOP
  ```

  because of testing the CX register twice on each pass.

- To avoid the double test, the following example can be used, but it is still not efficient:

  ```
  $DO
  $LEAVE  CXZ
    o
    o
    o
    DEC CX
  $ENDDO
  ```

  This Structure separates the decrementing and the testing of the result into two places, whereas in the first Structure the LOOP instruction does both functions.

Summary of where CXZ or NCXZ can be used:

STRUCTURE STATEMENT	NCXZ	CXZ
$IF CC	YES	NO
$IF CC,OR	NO	YES
$IF CC,AND	YES	NO
$LEAVE CC	NO	YES
$LEAVE CC,OR	NO	YES
$LEAVE CC,AND	YES	NO
$EXITIF CC	YES	NO
$EXITIF CC,,,NUL	NO	YES
$EXITIF CC,AND,,NUL	YES	NO
$EXITIF CC,OR,,NUL	NO	YES
$EXITIF CC,AND	YES	NO
$EXITIF CC,OR	NO	YES
$ENDLOOP CC	YES	NO
$ENDLOOP CC,AND	YES	NO
$ENDLOOP CC,OR	NO	YES

Logic Flow Structures

A fundamental concept of Structured Assembler is that the more obvious the flow of control of the program, the higher the reliability and maintainability of the code.

The flow of control in a Structured Assembler program is implemented by Logic Flow Structures. A Logic Flow Structure is a collection of Structure statements grouped together to do a procedure such as IF, IF ELSE, DO WHILE, DO UNTIL, DO COMPLEX, SEARCH UNTIL, SEARCH WHILE, and SEARCH COMPLEX. In Structured Assembler, the Structure statements expand into the Assembler statements required to implement each Logic Flow Structure.

It is easy to see the flow of control within Logic Flow Structures since they always are entered at the top and exited at the bottom. This simplifies the search for the flow of control.

Notes:

1. A Logic Flow Structure must have only *one entrance* and only *one exit* to qualify as a *Structure.*

2. A complete Logic Flow Structure is often called a *Process Block.*

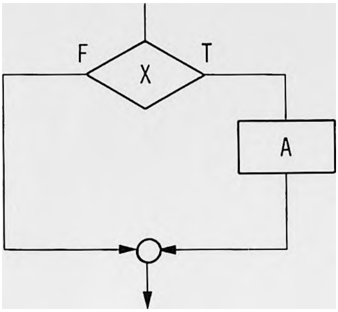
3. A Logic Flow Structure can be used as a single Process Block within another Logic Flow Structure; such usage is often called "nesting" of Logic Flow Structures.

Logic Flow Structure Examples

The following Logic Flow Structure examples show:

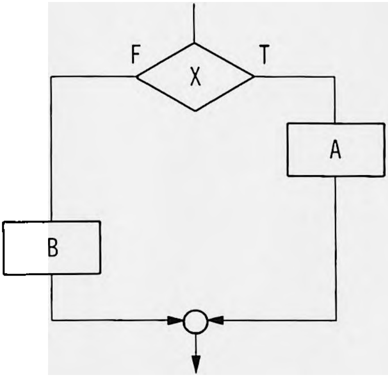
- Flowchart illustrating the logical program flow.
- Process Blocks represented as a box with an alphabetic character inside it.
- Structure statement usage.
- Jump instructions and labels generated by SALUT.
- X = Condition.
- A-E = Source program steps coded by user.
- L1-L4 = Labels generated by SALUT.

IF (simple IF, no ELSE)



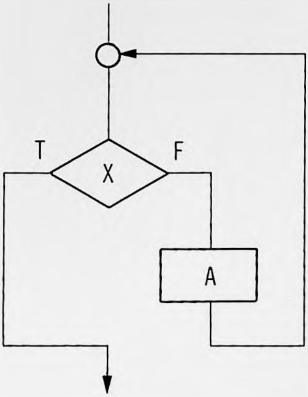
```
$IF X           JNX  L1

  A             A


$ENDIF        L1:
```

IF with ELSE



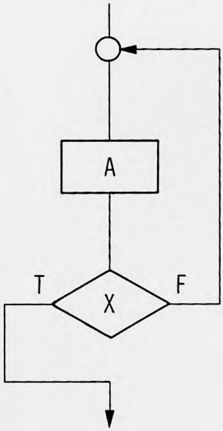
```
$IF X              JNX  L1

  A                A
$ELSE              JMP  L2
                 L1:
  B                B

 $ENDIF           L2:
```

DO WHILE (leading test)



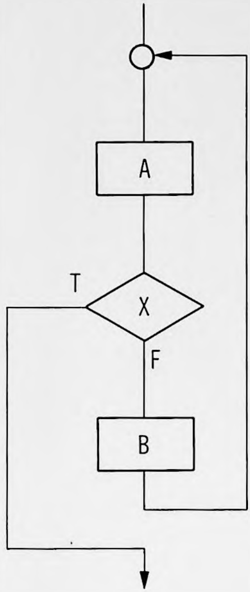
```
$DO            L1:

$LEAVE X          JX   L2

  A               A

$ENDDO            JMP  L1
               L2:
```

DO UNTIL (trailing test)



```
$DO            L1:

  A               A

$ENDDO X          JNX  L1
```

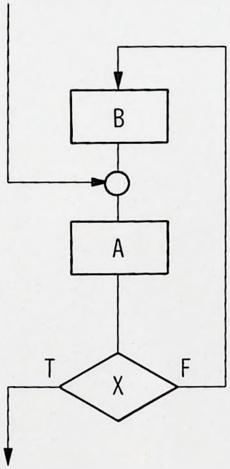
DO COMPLEX (middle exit)



```
$DO            L1:


  A               A



$LEAVE X          JX   L2



  B               B

$ENDDO            JMP  L1
               L2:
```

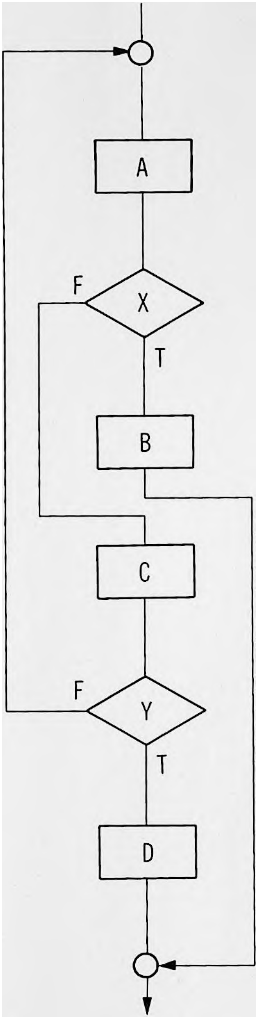
An alternate form of the COMPLEX DO (middle exit loop) uses the STRTDO command. This alternate form has the advantage of having one less unconditional jump execution for all non-first passes through the loop. The middle exit condition is moved to the physical end, and the beginning of the loop is effected by an initial jump into the physical middle of the body of the loop. Logically, this Flow Structure produces the same results as the flowchart on the preceding page.

DO COMPLEX (alternate form middle exit)



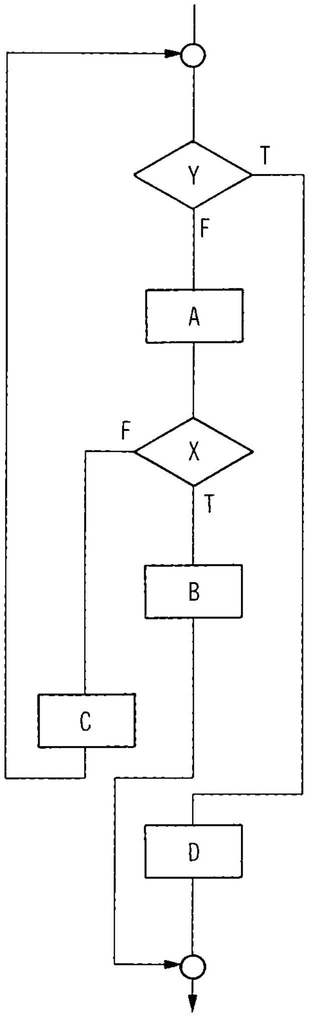
```
$DO COMPLEX        JMP  L2
                 L1:
  B                 B

$STRTDO          L2:

  A                 A



$ENDDO X           JNX  L1
```

SEARCH UNTIL (trailing test)



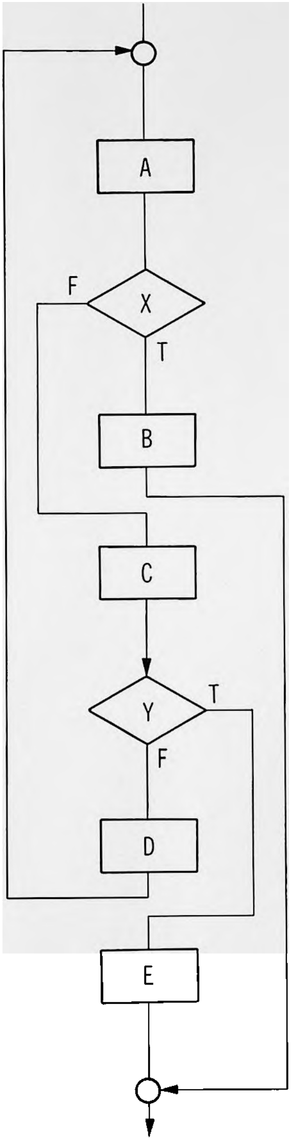
```
$SEARCH           L1:

  A                  A


$EXITIF X           JNX  L2


  B (success)       B

$ORELSE             JMP  L3
                  L2:
  C                 C


$ENDLOOP Y          JNY  L1


    D (failure)     D


$ENDSRCH          L3:
```

SEARCH WHILE (leading test)



```
$SEARCH        L1:

$LEAVE Y           JY   L2

  A                A

$EXITIF X          JNX  L3

  B (success)      B
$ORELSE            JMP  L4
               L3:
  C                C
$ENDLOOP           JMP  L1
               L2:
  D (failure)      D

$ENDSRCH       L4:
```

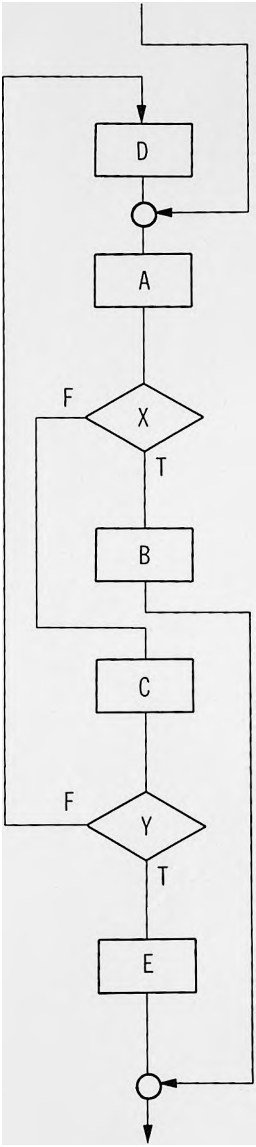
SEARCH COMPLEX (middle exit test)



```
$SEARCH          L1:


 A                  A


$EXITIF X           JNX   L2


  B (success)       B

 $ORELSE            JMP   L4
                 L2:

  C                 C


 $LEAVE Y           JY    L3


  D                 D

 $ENDLOOP           JMP   L1
                 L3:
  E (failure)       E


 $ENDSRCH        L4:
```

An alternate form of the COMPLEX SEARCH (middle exit loop) uses the STRTSRCH command. This alternate form has the advantage of having one less unconditional jump execution for all non-first passes through the loop. The middle exit condition is moved to the physical end, and the beginning of the loop is effected by an initial jump into the physical middle of the body of the loop. Logically, this Flow Structure produces the same results as the flowchart on the preceding page.

SEARCH COMPLEX (middle exit loop)



```
$SEARCH COMPLEX JMP  L3
             L1:


 D               D


$STRTSRCH    L3:


  A               A



$EXITIF X        JNX  L2



  B (success)     B


$ORELSE          JMP  L4
             L2:
  C               C


$ENDLOOP Y       JNY  L1




  E (failure)     E



$ENDSRCH     L4:
```

Structure Statement Usage

Structure statements are designed to help you develop an application of structured procedures. From two to as many as six of the Structure statements can be used to form the skeleton for each procedure of your program. The procedure can be as simple as *IF (condition) is true do the following process (A), otherwise Jump to next process (B)*, or as complex as a SEARCH that has five processing steps (A-E) with two intermediate condition tests. By using Structure statements to form the procedure skeleton, more efficient code is generated, while providing for Structured programming flow.

To begin your Structured Assembly source code, first enter all of the Structure statements that make up the desired complete structure. Second, enter the statements that make up the processing block(s) of that structure. For example, when coding the IF structure, first enter all three Structure statements, with comments, as in:

```
$IF (condition) ;IF PRICE EXCEEDS CASH ON HAND
$ELSE           ;SINCE PRICE DID NOT EXCEED
                ;CASH ON HAND
$ENDIF          ;PRICE EXCEEDED CASH ON HAND
```

Now insert the TRUE process block after $IF, and the FALSE process block after $ELSE.

Sample Development Process

Before attempting to code a complicated set of logic, lay out, in simple English statements, the steps to be performed. Show any decision making and all flow of control with ordinary English statements whose first word is the name of a Structured statement.

To properly indent this plain English logic, use SALUT, requesting just the format option.

> **Note:** Do not ask for the Pre-process option since this code is not assembler recognizable. See the "SALUT Sample Sessions" in Appendix D for details on how to preprocess, assemble, link and execute SALUT source code.

The following is a sample editing session that shows the design evolution of the SALEX1.SAL sample source file. This file resides on your assembler Diskette. As you follow this session, if you have any doubt about what logic path the flow of control is following, see the previous section "Logic Flow Structure Examples" for flowcharts of the logic patterns these Structure statements produce.

A program design starts off in a rather simple manner, consisting of only a few steps:

```
GET CONTROL FROM DOS
PERFORM THE PROCESS
RETURN TO DOS
```

In this description, the PERFORM THE PROCESS is obviously a rather high-level statement, and requires much more detail to define how that process is to be done. So we enlarge that one line by replacing it with several, adding a bit more detail:

```
GET CONTROL FROM DOS
CHECK THE PASSWORD
IF OK, PERFORM THE PROCESS
RETURN TO DOS
```

The new concept of requiring a password before doing the process has been added.

Remembering the need to use complete Structures each time, the IF needs to be better defined:

```
GET CONTROL FROM DOS
CHECK THE PASSWORD
IF OK,
  PERFORM THE PROCESS
ENDIF
RETURN TO DOS
```

This step introduces the idea of a conditional process. The ENDIF shows where the logic is to go if not OK. There is hole here, because of a need to show the *not OK* condition. An error process needs to be defined:

```
GET CONTROL FROM DOS
CHECK THE PASSWORD
IF OK,
  PERFORM THE PROCESS
ELSE
  SEND MESSAGE "INVALID PASSWORD"
ENDIF
RETURN TO DOS
```

The addition of the ELSE to the IF Structure shows what happens when the test for OK turns out to be false.

Now look at the statement CHECK THE PASSWORD. This seems to need more detail also, so replace that one line with:

```
READ PASSWORD FROM KEYBOARD
COMPARE RESPONSE WITH CORRECT PASSWORD
```

Now the logic is:

```
GET CONTROL FROM DOS
READ PASSWORD FROM KEYBOARD
COMPARE RESPONSE WITH CORRECT PASSWORD
IF OK,
  PERFORM THE PROCESS
ELSE
  SEND MESSAGE "INVALID PASSWORD"
ENDIF
RETURN TO DOS
```

It is poor form to blindly read from the keyboard without a prompt asking the user for a response, so we add the prompt:

```
GET CONTROL FROM DOS
DISPLAY PROMPT "ENTER PASSWORD" TO CONSOLE
READ PASSWORD FROM KEYBOARD
COMPARE RESPONSE WITH CORRECT PASSWORD
IF OK,
  PERFORM THE PROCESS
ELSE
  SEND MESSAGE "INVALID PASSWORD"
ENDIF
RETURN TO DOS
```

This may seem to be enough, but we all know what it is like to be a fumble fingers on the keyboard. We need to be patient and not kick the user off for incorrectly entering the password. We allow the user to try again. The concept of doing something again, or in a repeated process, is called a loop, which is usually handled with the DO-ENDDO Structure. This calls for some rethinking of our previous design:

```
GET CONTROL FROM DOS
DO
  DISPLAY PROMPT "ENTER PASSWORD" TO CONSOLE
  READ PASSWORD FROM KEYBOARD
  COMPARE RESPONSE WITH CORRECT PASSWORD
ENDDO
IF OK,
  PERFORM THE PROCESS
ELSE
  SEND MESSAGE "INVALID PASSWORD"
ENDIF
RETURN TO DOS
```

Now that is not right, because there is no exit from the loop. ENDDO goes back to the DO and repeats the prompt and its response. Something is in the wrong place. Let us move the ENDDO to include the exit test:

```
GET CONTROL FROM DOS
DO
  DISPLAY PROMPT "ENTER PASSWORD" TO CONSOLE
  READ PASSWORD FROM KEYBOARD
  COMPARE RESPONSE WITH CORRECT PASSWORD
  IF OK,
    PERFORM THE PROCESS
  ELSE
    SEND MESSAGE "INVALID PASSWORD"
  ENDIF
ENDDO
RETURN TO DOS
```

That may be better, yet the test still does not exit from the loop. There must be a way to leave (exit) this repeated process:

```
GET CONTROL FROM DOS
DO
  DISPLAY PROMPT "ENTER PASSWORD" TO CONSOLE
  READ PASSWORD FROM KEYBOARD
  COMPARE RESPONSE WITH CORRECT PASSWORD
LEAVE IF OK
  SEND MESSAGE "INVALID PASSWORD"
ENDDO
PERFORM THE PROCESS
RETURN TO DOS
```

The IF ELSE ENDIF has become the LEAVE from the loop. When the LEAVE condition is satisfied, (the password is OK), control goes to the next line after the ENDDO, outside the loop, where the OK response causes the PROCESS TO BE PERFORMED. The failure of an invalid password does not exit the loop, but falls through the LEAVE to the INVALID PASSWORD message. The ENDDO goes back to DO so we can try again.

It may be acceptable to leave things as they are, but what if this user really has no business with this application? After all, that is what passwords are all about. The logic above keeps asking for the password until the user gets it right or dies of old age. We do not want the user to keep trying, so let us set some practical limit, say three chances, and if the user still has not gotten it right, then kick the user out. The three strikes and you're out idea gives us another reason to exit from that DO ENDDO loop, which now becomes:

```
GET CONTROL FROM DOS
SET LOOP COUNTER TO THREE
DO
  DISPLAY PROMPT "ENTER PASSWORD" TO CONSOLE
  READ PASSWORD FROM KEYBOARD
  COMPARE RESPONSE WITH CORRECT PASSWORD
LEAVE IF OK
  SEND MESSAGE "INVALID PASSWORD"
  DECREMENT LOOP COUNTER BY ONE
LEAVE IF COUNT IS ZERO
ENDDO
PERFORM THE PROCESS
RETURN TO DOS
```

That allows the count of three, but now we have a problem. If the wrong password is entered enough times, the program does "forgive" and PERFORM THE PROCESS anyway! That is not what we intended. There are two ways out of that loop, and only the "OK" way is supposed to allow for the PERFORM THE PROCESS step, so let us add a test:

```
GET CONTROL FROM DOS
SET LOOP COUNTER TO THREE
DO
  DISPLAY PROMPT "ENTER PASSWORD" TO CONSOLE
  READ PASSWORD FROM KEYBOARD
  COMPARE RESPONSE WITH CORRECT PASSWORD
LEAVE IF OK
  SEND MESSAGE "INVALID PASSWORD"
  DECREMENT LOOP COUNTER BY ONE
LEAVE IF COUNT IS ZERO
ENDDO
COMPARE RESPONSE WITH CORRECT PASSWORD
IF OK
  PERFORM THE PROCESS
ENDIF
RETURN TO DOS
```

Perhaps we should tell the user that he, or she, was kicked out, so add a false leg to the IF Logic at the end:

```
GET CONTROL FROM DOS
SET LOOP COUNTER TO THREE
DO
  DISPLAY PROMPT "ENTER PASSWORD" TO CONSOLE
  READ PASSWORD FROM KEYBOARD
  COMPARE RESPONSE WITH CORRECT PASSWORD
LEAVE IF OK
  SEND MESSAGE "INVALID PASSWORD"
  DECREMENT LOOP COUNTER BY ONE
LEAVE IF COUNT IS ZERO
ENDDO
COMPARE RESPONSE WITH CORRECT PASSWORD
IF OK
  PERFORM THE PROCESS
ELSE
  SEND MESSAGE "THREE STRIKES, YOU'RE OUT!"
ENDIF
RETURN TO DOS
```

Now if the user misses the password three times, the LEAVE IF COUNT IS ZERO takes them out, but the IF OK fails, thus skipping over the PERFORM THE PROCESS, which is what we wanted. Look closely! We have some repetition. Do you see the COMPARE RESPONSE WITH CORRECT PASSWORD and the following check for OK? Repeating the test is inefficient. We just did that test to see if we could exit the loop, and now that we exited the loop, we are repeating that same test just to find out which reason took us out of the loop. We have two ways out of that loop, and it does make a difference which reason caused the exit. *That* is how to recognize a need for *SEARCH*.

Rewrite the loop, using SEARCH for DO, ENDLOOP for ENDDO, and put one exit (the OK one) on EXITIF and the other exit (three strikes, you're out) on the ENDLOOP. The SEARCH Logic Structure allows for two unique processing blocks, one called SUCCESS, and the other called FAILURE, depending on whether the exit from the loop is the result of finding what you were looking for. We are looking for the correct response matching the desired password. Replace DO with SEARCH, LEAVE IF OK with EXITIF OK, ENDDO with ENDLOOP. Now we have:

```
GET CONTROL FROM DOS
SET LOOP COUNTER TO THREE
SEARCH
  DISPLAY PROMPT "ENTER PASSWORD" TO CONSOLE
  READ PASSWORD FROM KEYBOARD
  COMPARE RESPONSE WITH CORRECT PASSWORD
EXITIF OK
  SEND MESSAGE "INVALID PASSWORD"
  DECREMENT LOOP COUNTER BY ONE
LEAVE IF COUNT IS ZERO
ENDLOOP
COMPARE RESPONSE WITH CORRECT PASSWORD
IF OK
  PERFORM THE PROCESS
ELSE
  SEND MESSAGE "THREE STRIKES, YOU'RE OUT!"
ENDIF
RETURN TO DOS
```

Which needs repairing. The success block, PERFORM THE PROCESS, goes just after the EXITIF. The success block is to be ended by the ORELSE. Getting rid of the duplicate test for OK password leaves us with:

```
GET CONTROL FROM DOS
SET LOOP COUNTER TO THREE
SEARCH
  DISPLAY PROMPT "ENTER PASSWORD" TO CONSOLE
  READ PASSWORD FROM KEYBOARD
  COMPARE RESPONSE WITH CORRECT PASSWORD
EXITIF OK
  PERFORM THE PROCESS
ORELSE
  SEND MESSAGE "INVALID PASSWORD"
  DECREMENT LOOP COUNTER BY ONE
LEAVE IF COUNT IS ZERO
ENDLOOP
  SEND MESSAGE "THREE STRIKES, YOU'RE OUT!"
RETURN TO DOS
```

The only thing missing is the ENDSRCH.

> **Note:** Every SEARCH Structure has 5 elements: SEARCH, EXITIF, ORELSE, ENDLOOP, and ENDSRCH, respectively.

Add the missing piece at the end of the *failure* block. Just as ORELSE indicated the end of the *success* block, ENDSRCH shows the end of the *failure* block, giving us:

```
GET CONTROL FROM DOS
SET LOOP COUNTER TO THREE
SEARCH
  DISPLAY PROMPT "ENTER PASSWORD" TO CONSOLE
  READ PASSWORD FROM KEYBOARD
  COMPARE RESPONSE WITH CORRECT PASSWORD
EXITIF OK
  PERFORM THE PROCESS
ORELSE
  SEND MESSAGE "INVALID PASSWORD"
  DECREMENT LOOP COUNTER BY ONE
LEAVE IF COUNT IS ZERO
ENDLOOP
  SEND MESSAGE "THREE STRIKES, YOU'RE OUT!"
ENDSRCH
RETURN TO DOS
```

The OK test needs more refinement. A correct password has two features, it must have the right characters, and be the right length. Adding EXITIF makes a two-part test to meet the two requirements, which is a pair of EXITIFs logically connected with an AND or OR. Now we have:

```
GET CONTROL FROM DOS
SET LOOP COUNTER TO THREE
SEARCH
  DISPLAY PROMPT "ENTER PASSWORD" TO CONSOLE
  READ PASSWORD FROM KEYBOARD
  COMPARE LENGTH OF RESPONSE WITH
          LENGTH OF PASSWORD
EXITIF LENGTH OF PASSWORD IS OK, AND
  COMPARE RESPONSE WITH CORRECT PASSWORD
EXITIF TEXT COMPARES OK
  PERFORM THE PROCESS
ORELSE
  SEND MESSAGE "INVALID PASSWORD"
  DECREMENT LOOP COUNTER BY ONE
LEAVE IF COUNT IS ZERO
ENDLOOP
  SEND MESSAGE "THREE STRIKES, YOU'RE OUT!"
ENDSRCH
RETURN TO DOS
```

> **Note:** The *AND* in the first EXITIF means both tests must be OK before the *success* process block gets control.

If either test fails, (control goes to the ORELSE) we have an error response condition.

An excellent way to do a loop for a particular number of times is:

- Put the loop count in the count register (CX).
- Combine the DECREMENT LOOP COUNT BY ONE, the LEAVE IF COUNT IS ZERO, and the ENDLOOP by using the “loop” condition.

Now we have:

```
GET CONTROL FROM DOS
SET LOOP COUNTER REGISTER (CX) TO THREE
SEARCH
  DISPLAY PROMPT "ENTER PASSWORD" TO CONSOLE
  READ PASSWORD FROM KEYBOARD
  COMPARE LENGTH OF RESPONSE WITH
          LENGTH OF PASSWORD
EXITIF LENGTH OF PASSWORD IS OK, AND
  COMPARE RESPONSE WITH CORRECT PASSWORD
EXITIF TEXT COMPARES OK
  PERFORM THE PROCESS
ORELSE
  SEND MESSAGE "INVALID PASSWORD"
ENDLOOP LOOP
  SEND MESSAGE "THREE STRIKES, YOU'RE OUT!"
ENDSRCH
RETURN TO DOS
```

Which is simpler.

For a verification pass through SALUT, change the flow of control verbs to Structure statements by adding the $ symbol, and change each *condition* to one word by using underscores:

```
GET CONTROL FROM DOS
SET LOOP COUNTER REGISTER (CX) TO THREE
$SEARCH
  DISPLAY PROMPT "ENTER PASSWORD" TO CONSOLE
  READ PASSWORD FROM KEYBOARD
  COMPARE LENGTH OF RESPONSE WITH
          LENGTH OF PASSWORD
$EXITIF LENGTH_OF_PASSWORD_IS_OK, AND
  COMPARE RESPONSE WITH CORRECT PASSWORD
$EXITIF TEXT_COMPARES_OK
  PERFORM THE PROCESS
$ORELSE
  SEND MESSAGE "INVALID PASSWORD"
$ENDLOOP LOOP
  SEND MESSAGE "THREE STRIKES, YOU'RE OUT!"
$ENDSRCH
RETURN TO DOS
```

Request SALUT for just a format operation without the pre-process operation to check for errors. The result is a natural language description of the logic, and the flow of control, with automated indentation:

```
GET     CONTROL FROM DOS
SET     LOOP COUNTER REGISTER (CX) TO THREE
$SEARCH
    DISPLAY PROMPT "ENTER PASSWORD" TO CONSOLE
    READ    PASSWORD FROM KEYBOARD
    COMPARE LENGTH OF RESPONSE
            WITH LENGTH OF PASSWORD
$EXITIF LENGTH_OF_PASSWORD_IS_OK,AND
    COMPARE RESPONSE WITH CORRECT PASSWORD
$EXITIF TEXT_COMPARES_OK
    PERFORM THE PROCESS
$ORELSE
    SEND    MESSAGE "INVALID PASSWORD"
$ENDLOOP LOOP
    SEND    MESSAGE "THREE STRIKES, YOU'RE OUT!"
$ENDSRCH
RETURN  TO DOS
```

The indentation is done with default column settings; define your own if you have other preferences.

Now we see how to design the logic of our programs and remain in control of that logic.

> **Note:** This approach was used to develop the SALUT example source program for a similar case in Appendix D.

Starting SALUT

The SALUT program can be run in prompt mode or in non-interactive mode.

When specifying the input file, the default extension is .SAL, but it can be overridden. The extension of the formatted and pre-processed filespecs default to .SAL and .ASM, unless otherwise specified. They can be specified as only a device name. If the device name is A, B, C, etc., then this drive name is used with the input filename and the specified or default extension to produce the filespec. If any other device name, such as SCRN or LPT1 is specified, it is used as the complete filespec. The DOS default drive is assumed if the filename does not contain the drive name.

If the input file and the formatted output file are both using the same filespec and if both are on the same drive, the input file is first renamed to become INFILNAM.BAK, so the formatted output file can become the replacement (new master) file. If any file called INFILNAM.BAK file is already on the disk, left from a previous execution of SALUT, it is first deleted.

Non-Interactive Mode

The non-interactive mode is where the complete SALUT command line with options is entered at the DOS prompt, or in a batch file. The format of the SALUT command line is:

```
SALUT in-file[,[out1-file/NUL][,out2-file/NUL]][(i,o,r,c)][/R]
```

> **Note:** Pathnames can be added before any of the 3 data filenames. The default path is the current subdirectory.

To specify the original master source, the input filespec (in-file) is required. If the input file resides on the DOS default drive and has a file extension of .SAL, then the file name instead of filespec can be used.

If requested, SALUT builds a new master source file as a properly indented, formatted output file (out1-file) using either default or user-supplied columns. Unless NUL is specified, SALUT builds a formatted output file ready for another SALUT operation. Use the desired filespec, filename.ext for DOS default drive, or just file-name and SALUT builds the output file with file-name.SAL on the DOS default drive. Use NUL if no formatted output file is desired at this time. This file is similar to the input file in content, but formatted in columns-wise for readability. The (i,o,r,c) sub-field can be used to override the default column settings. Refer to the following options section for more detail.

Unless NUL is specified, SALUT builds a pre-processed output file (out2-file/NUL) ready for assembly. Use the desired filespec, filename.ext for DOS default drive, or just file-name and SALUT builds the output file with file-name.ASM on the DOS default drive. Use NUL if no pre-processed output file is desired.

The /R can be used to replace tabs (use blanks instead of tab characters) from output files. This option produces larger output files but eliminates potential printing problems. Determine whether your editor inserts tabs when saving files and include this option if it does.

Non-Interactive Mode Examples

```
SALUT SOURCE
```

The SOURCE.SAL file on the DOS default drive is renamed to SOURCE.BAK. A properly indented new master source file SOURCE.SAL is created on the DOS default drive. A temporary file SOURCE.ASM is also created and assigned to the default drive. SOURCE.ASM not only shows the indenting, but also the results of converting the Structure statements to assembler instructions and labels. After executing the assembler to process this file, delete SOURCE.ASM to avoid accidental editing of a temporary source file.

```
SALUT B:SOURCE
```

Assuming B is not the default drive, the file B:SOURCE.SAL remains unchanged. The new SOURCE.SAL file is created on the DOS default drive. SOURCE.SAL is created with nested structures properly indented to become the new master source file. SOURCE.ASM is created as in the previous example.

```
SALUT SCOTT,NUL
```

This example does not rename the source file or create a new master source file. A SCOTT.ASM file with expanded Structure statements is created; however, the indentation remains the same as the input file, SCOTT.SAL.

```
SALUT JONES.PDL,,NUL(2,4,6
```

The JONES.ASM file is not created. The JONES.SAL file is created, and properly indented according to the new column specification entered. Each nested structure is indented only two columns. The op-code is shifted to column 4; the operand to column 6; the remarks by default go to column 41. When using new column specification, not all fields need to be specified if the remaining default values are acceptable.

Prompt Mode

SALUT prompts the user for commands if none are specified on the DOS prompt line.

```
SALUT
```

The above example entry causes SALUT to load, clear screen, print the option list. The user then responds with the desired options. After accepting options, SALUT requests the filespecs for input and output files. A different drive and pathname may be included with each of the three filespecs.

SALUT accepts single or multiple option entries in either uppercase or lowercase. Multiple option entries are made as a string of characters, in any order, without separators. SALUT returns to the option menu on completion (thus allowing for the processing of several files with one load of SALUT) unless the Q option is selected along with other options. If option Q is also selected, SALUT returns to DOS on completion of other options selected at that time.

Options

Q	Quit. SALUT ends after doing the other requested options.
P	Pre-process. SALUT Structure statements are expanded. The pre-process function produces a file whose default name is the same as the source but with the extension

of .ASM. This temporary file passes to the assembler and should then be deleted to avoid accidental editing of this non-source file.

F	Format. Create a new master-file whose default extension is .SAL. If the new master-file is defined as being in the same subdirectory as the old master source, then the old master source extension is renamed to .BAK. The source is indented according to the column specifications. All opcodes beginning with $ are converted to uppercase. The code within each Structure is indented by four columns, while maintaining the same opcode to operand spacing. If the actual data exceeds the size of the standard fields, data is placed as close to the standard locations as possible. All lines with the remark indicator (;) in column 1 remains unchanged. If the default columns are used, the formatter places labels in column 1, opcodes in column 9, operands in column 17, and remarks in column 41.

The default column specification can be overridden by entering new column values as sub-fields with the parentheses (i,o,r,c). The sub-fields are:

i	Structure indent, valid range is 0-32,default is 4.
o	Opcode column, valid range is 2-33, default is 9.
r	Operand column, valid range is opcode column plus 2 through 65, default is 17.
c	Remark column, valid range is 1-97, default is 41.

The column setting parameters must be enclosed in parentheses, and can appear anywhere within the option string. The ending parenthesis can be omitted if the column parameters are at the end of the option string. The new column settings remain until changed again or SALUT is reloaded. Commas are required between sub-fields or in place of sub-fields not changed. However no commas are required following the new sub-field. For example, if only the *r* column parameter needs changing then the format is (,,*r*).

If the sub-field specified is not within the valid range, one of these error messages is produced:

- Invalid Structure indent
- Invalid opcode column
- Invalid operand column
- Invalid remark column

R	Replace tabs. Tabs are replaced with blanks everywhere outside quotes. This results in much larger output and pre-process files, but it eliminates problems when printing the files.

Example response:

```
PFQ (2
```

This response tells SALUT to format a new master-file using only a 2-column indentation to show nested structures, and produce an .ASM file for the assembler. SALUT returns to DOS on completion.

Error Messages

When SALUT detects an error, a “SALUTERR” (with error message text) command line is inserted in both the formatted output and in the pre-processed output. When editing to fix the error, it is not necessary to delete this SALUTERR error line. If the .ASM file containing the SALUTERR command is passed on to the assembler, it flags that line as containing an undefined op-code, thus adding it to its count of errors.

The text of the SALUTERR statement consists of one of the following:

- Structure mismatch
- Invalid Structure statement continuation
- Invalid condition
- No Structure active
- Invalid parameter
- Required parameter missing

Terminating Errors

SALUT is written in the BASIC compiler language. BASIC detects and passes certain errors to SALUT for proper handling. The errors are indicated by these messages:

- Integer overflow
- Device not ready
- Invalid filespec
- File not found
- File already open
- File already exists
- Disk full
- Too many files or bad filespec
- Diskette write-protected
- Disk media error
- “Error”;ernum;“in line”;linenum

Chapter 5. Macro Assembler

Contents

Introduction **5-3**
  Pass 1 and Pass 2 Differences 5-3
  Phase Errors 5-5
  Forward Reference 5-6
  Device Identifications 5-6

The Assembly Step **5-8**
  Before You Begin 5-8
  Parameters (/parms) 5-10
  Option 1 — Console Responses 5-12
  Option 2 — Command Line Responses . . . 5-14
  Option 3 — Batch Command 5-17
  Assembler Termination 5-17

Viewing Assembler Files **5-18**

Create a Cross Reference **5-19**

Eliminate All Errors **5-20**

The Linker Step **5-20**

Viewing Linker Files **5-21**

Executing Your Program **5-22**

Use DEBUG **5-23**
  1. Reversed Order of Bytes 5-23
  2. IP or SEG Displacement 5-24
  3. Fields Modified by LINK or LOADER . 5-25

Professional Debug Facility **5-25**

Introduction

Your IBM Personal Computer Macro Assembler is a two-pass assembler that reads your source program twice. The first pass is used to define what the relative offset is for each line of source. The second pass generates the listing, object, and CREF files. Even though these are two different passes, the assembler executes the same code for each with minor differences between them.

Pass 1 and Pass 2 Differences

Pass 1:

- Only certain types of errors are displayed (errors involving items that must be defined on pass 1; such as, bad .RADIX value).
- No object code is produced.
- If undefined items are found, assumptions are made about what the symbol is so that the correct number of bytes is generated on pass 1.
- No listing is produced. (The /D parameter produces a list on both passes.)

Pass 2:

The assembler uses the values defined in pass 1 to generate the output. When the definition of a symbol is found on pass 2, the pass 2 value is checked against the pass 1 value (which is in the symbol table). If they are different, a phase error is generated.

Because pass 1 must keep correct track of the relative offset, some references must be known on pass 1. If they are not, the relative offset will not be correct.

The following references must be known on pass 1:

- **IF/IFE *expression*.** If *expression* is not known on pass 1, the assembler does not know which block of the conditional to assemble; therefore, a phase error is guaranteed on pass 2.
- ***expression*** DUP(...). This explicitly changes the relative offset, so *expression* must be known. The (...) does not have to be known because it does not affect the number of bytes generated.
- **RADIX *expression*.** Because this changes the base, constants can be different, which can cause the assembler to improperly evaluate the IFs or DUPs.

The IF1 and IF2 pseudo-ops can also be used to tell you if the assembler is in pass 1 or pass 2. Examples of IF1/IF2 use are:

- If there are conditionals that determine the version of the program, %OUT can be used under IF1/IF2 to display which version is being assembled.
- Some programs have many MACROs, but they are only defined once (could be in a separate file and assembled into the current source file by using the INCLUDE pseudo-op). To save time redefining these MACROs in pass 2, when they have not changed, an IF1 can be placed before the definitions that are followed by an ENDIF.

Phase Errors

Phase errors indicate that the assembler found a different value for a LABEL, VARIABLE or PROC on pass 2 than it did on pass 1. Some of the causes are:

- Improper use of the IF1/IF2 (different number of bytes on each pass).
- Forward reference on pass 1 of STRUC (no bytes on pass 1 since undefined; generates STRUC on pass 2).
- Forward reference in an instruction that violates the assembler assumptions.
- Results of pass 1 detected an error.

An error can be between the last label before the one with the error and the label. After checking the code, the error may be obvious. If it is not, you can use the /D parameter and compare the relative offset in both listings. This should isolate the error. The pass 1 listing looks just like the pass 2 listing except:

- Errors can occur because of forward references.
- Different operand codes can be generated on pass 1.

> **Note:** Errors generated on pass 1 that are normally displayed are not counted as errors in the final tally.

Forward Reference

Placing variables in the beginning of source code enables the assembler to recognize the attributes (type, segment, offset) of these operands before they are used in following code. This permits the assembler to produce more efficient code, by generating shorter instructions. References to data always use a 2-byte offset; however, transfers of control can vary from 1, 2, or 4 bytes depending on the context of usage and specification. Registers can not be forward-referenced. A forward reference later determined to be defined as a register, generates an error.

Device Identifications

If you want to override the implied DOS default diskette drive, you can specify the device identification for your output devices.

The device names and designated identifications (IDs) are:

- Diskette and fixed disk drives — A, B, C, D, etc...
- Display screen — CON (buffered output), USER (non-buffered)
- RS-232 (serial port) — COM1 (buffered output), or LINE (non-buffered).

To generate a printed copy:

- Direct the listing to USER and press the Ctrl-PrtSc keys to echo (copy) the screen's contents to the printer,

  or,

- Direct the listing to a diskette file, press the Ctrl-PrtSc keys, and enter the DOS TYPE command.

  or,

- Use the DOS 2.0 (or later) Print command to send the .LST file to the printer.

A printed listing can be generated by entering "LPT1:", but this method is not recommended because the output can contain tab characters that the printer may not properly interpret. Do not copy the listing file to the printer since this method may also produce problems with the TAB.

You can suspend program execution by pressing the Ctrl and Num Lock keys. Press any character to resume. When a large amount of output is generated, suspension is necessary to read the screen.

The Assembly Step

Before You Begin

This list provides information and suggestions for running your assembler.

- Make sure the files you use for assembly are on the appropriate disks. (Use the DOS DIR command to identify the files on the diskette.)
- Make sure you have enough free space on the disk to contain your files and the data files that are produced by the assembler.
- References to starting the assembler show the choice of assemblers by using the notation: [M]ASM. When possible, use the Macro Assembler, MASM, instead of the Small Assembler, ASM. In those cases where available memory is limited (you receive an error message, "OUT OF MEMORY"), you can use ASM, which takes up less memory. Be aware, however, that ASM does not support some of the features that MASM supports (see Chapter 1 "General Information") and cannot be used if your application requires any of these features to execute properly.
- You can stop the execution of the assembler before its normal exit by pressing the Ctrl and Break keys together.
- All "prompts" are requests for information from the user and require responses before the assembler goes to the next prompt or executes your command.

- The name shown within the brackets is the *default* filename extension that the assembler uses if you do not choose a filename extension of your own.

- Although the assembler supplies a default filename extension whenever you do not supply one, all extensions can be overridden by explicitly specifying the filename with the new extension.

- The NUL filename indicates no file is created unless the user defines a filename. You can also specify "NUL" to override the creation of the object (.OBJ) file.

- For dual-drive systems, the default diskette drive is the DOS default drive; it can be overridden by including the drive ID as part of the filespec.

- For DOS version 2.0, the default path is the current directory. Other sub-directories can be referenced by indicating the path name before the filename. If the file is assigned to a path other than the default path, that path must be specified.

- Parameters (*/parms*) are never prompted; they can be added to the end of the command line or to any file specification given in response to a prompt (see the "Parameters" section earlier in this chapter for a list of available parameters).

Parameters (/parms)

A parameter is a letter appended to the command string (option 2) or any prompt (option 1). A parameter must be preceded by the slash (/). The letter specifies an optional task that must be carried out during assembly. You can use more than one parameter in any order; however, *each* parameter must be preceded by the slash (/). The available parameters and their meanings are:

- /A—The assembler arranges the source's segments in alphabetical order (see /S parameter below).

  Note: An alphabetic arrangement is the default arrangement of this assembler and the original assembler (version 1.0).

- /D—This parameter produces a listing on both assembler passes (pass 1 and pass 2). The listing provides information that allows you to resolve phase errors between pass 1 and pass 2. The listing pseudo-op, .XLIST, is ignored.

- /E—Assembles for 8087 and generates floating point constants in 8087 format (same as /R). This parameter also generates external linkage to the 8087 Emulation package provided in the FORTRAN and PASCAL libraries.

  Note: If neither /R or /E is specified, and if your source does not use the .8087 pseudo-op, floating point constants are defined in the Microsoft format (as used by the BASIC interpreter).

- /N—Suppresses the generation of symbol tables at the end of listings.

- /O—The machine language generated and the offsets are shown in Octal notation.

- /R—Assembles for 8087 and generates constants in 8087 format. This parameter performs the same function as the .8087 pseudo-op. Proper execution of the resulting code requires the presence of the 8087 Numeric Co-Processor hardware option.

- /S—The assembler leaves the source segments in their specified order of occurrence (as they appear in the source, see /A parameter above).

- /X—Suppresses the listing of false conditions. This parameter is used with the pseudo operations: .SFCOND, .LFCOND, and .TFCOND. See Chapter 2 "Pseudo Operations" in the *Macro Assembler Reference* book for a description of these pseudo-ops.

For example:

```
ASM MODULE,,/O/N;
```

In the listing, the machine language is in Octal, and no symbol table is printed.

Option 1 — Console Responses

From the keyboard, type:

```
ASM
           or
MASM
```

The assembler is loaded into memory and displays a series of four prompts, one at a time, to which you must enter the requested responses. The four prompts and their corresponding response formats are:

PROMPT	RESPONSES
Source filename [.ASM]:	[*d*:][*path*]*filename*[.*ext*][/*parm*]...[;]
Object filename [filename.OBJ]:	[*d*:][*path*][*filename*[.*ext*]][/*parm*]...[;]
Source listing [NUL.LST]:	[*d*:][*path*][*filename*[.*ext*]][/*parm*]...[;]
Cross reference [NUL.CRF]:	[*d*:][*path*][*filename*[.*ext*]][/*parm*]...[;]

PROMPT	DEFINITION
Source filename	Is the name of the source (input) file.
Object filename	Is the name of the machine-readable object file produced by the assembler.
Source listing	Is the source file listing produced by the assembler.
Cross reference	Is the symbol file produced by the assembler and used by the CREF utility to produce a cross reference listing. If this file is requested, the listing file will have a column of line numbers along the left margin for cross referencing purposes, and the rest of the listing will become shifted to the right by 8 print positions.

RESPONSES	DEFINITION
d:	Is the drive containing the file. If the drive is not specified, the default drive is used. For example: A, B, C, etc...
path	Is the path used to locate the file's directory. If the path is not specified, the default path is used. (See the "Using Tree-Structured Directories" chapter in the DOS book for details.)
filename	Is the name of the file.
.ext	Is the filename extension of the file. (See the "Default Filename Extensions" section earlier in this chapter for details.)
/parm	Is the parameters you specify. (See the "Parameters" section earlier in this chapter for details.)
;	Is the ending response, the remaining responses are all assumed to be the default.

The "Source filename [.ASM]:" prompt is the only prompt that requires a response. All subsequent prompts show the default used if you respond by pressing Enter. After entering your source filename, before pressing enter, you can continue the response with a comma and the answer to what would be the next prompt, without having to wait for that prompt. If you end any response with a semicolon (;), the remaining responses are all assumed to be the default. Processing begins immediately with no further prompting.

When you have entered the last filename, the assembler begins processing. If the assembler finds any errors, it displays the errors on the screen and in the listing file.

Option 2 — Command Line Responses

From the keyboard, enter:

```
[M]ASM source,object,list,cross-ref [/parm]...;
```

source	Is the name of your source file.
object	Is the name you want to give to the object file.
list	Is the name you want to give to the assembler listing file.
cross-ref	Is the name you want to give to the cross reference symbol file.
parm	Is an optional assembler parameter. Each parameter must begin with a slash (/).
;	Is the ending response, the remaining responses are all assumed to be the default.

The assembler is loaded and immediately produces the files indicated by the command field.

When you use the command line, the prompts described in Option 1 are not displayed if you specify an entry for all four files or end the command line with a semicolon.

If you do not specify all four files and do not end your command line entries with a semicolon, the assembler prompts for the remaining unspecified files.

Each prompt displays a default that you can accept by pressing the Enter key, or override by entering a filename or device name. However, if you enter an incomplete list and terminate the command line with a final semicolon, the unspecified files are defaulted without further prompting.

Examples:

From the keyboard, enter:

[M]ASM

A prompt shows the default extension of .ASM. After entering the response, a prompt shows the default of *filename*.OBJ. After entering the response, a prompt shows the default of NUL.LST. After entering the response, a prompt shows the default of NUL.CRF.

[M]ASM module

Source is module.ASM. A prompt shows the default of module.OBJ. After entering the response, a prompt shows the default of NUL.LST. After entering the response, a prompt shows the default of NUL.CRF.

[M]ASM module;

Because the semicolon is added, no further prompts display. The source of module.ASM is assembled; the object is put into module.OBJ; no listing or cross reference file is produced.

[M]ASM module,,;

This is similar to the above example, except the listing is produced in module.LST.

[M]ASM module,,,;

This is similar to the preceding examples, except the cross reference file, module.CRF, is also produced.

[M]ASM module,,,

Using the same example, but without the semicolon, module.ASM is assembled, the object is produced in module.OBJ, a listing is module.LST, but a prompt is given with the default of module.CRF.

[M]ASM module,NUL,;

No object is produced. The listing is in module.LST. No cross reference is produced and no further prompts are displayed.

Note: In the above four examples, the specification of a comma before a field where default is normally "NUL" changes that default to "module".

Option 3 — Batch Command

Use an editor to create the following sample batch command file (ASSEM.BAT) to display the listing and cross reference table:

```
A:MODE LPT1:132
B:
A:MASM %1,NUL,,;
IF ERRORLEVEL 1 GOTO :QUIT
TYPE %1.LST
ERASE %1.LST
A:CREF %1;
TYPE %1.REF
ERASE %1.REF
ERASE %1.CRF
:QUIT
```

Note: In this example, assume drive B has source and space for a listing, and drive A has DOS, the assembler and CREF.

To use this Batch option, put the printer (if you have a printer) in the echo mode by pressing Ctrl-PrtSc to capture hard copy of whatever is displayed on the screen. Then enter:

```
ASSEM MODULE
```

Note: See the IBM Personal Computer *Disk Operating System (DOS)* book for a detailed description of the Batch command facility that can be used to automatically start the assembler.

Assembler Termination

When the assembler finishes running, control is returned to DOS. An ERRORLEVEL Return code is set:

0	No errors
1	Any error

Each assembly error for the Macro Assembler (MASM) results in an error message and the statement causing the error to be displayed on the screen. For the Small Assembler (ASM), only the error code is displayed. (If the list field does not specify USER or CON:, the screen contains only errors.)

This completes the assembly step. You have named a source file, created an object file (object module), and (optionally) created source and reference listings for one module. If your application is composed of more than one module, repeat the assembly process for each module using any of the three options described in this chapter.

Viewing Assembler Files

Use the DOS DIR command to display the files produced by the assembler. The DOS TYPE command displays the contents of the .LST file. If you have a printer and want a printed copy, first press the Ctrl-PrtSc keys then enter the DOS TYPE command. Press the Ctrl-PrtSc keys (again) to stop the printer. (Each Ctrl-PrtSc press switches into or out of echo-to-print mode.) To read the screen, press Ctrl-Num Lock to stop the scrolling action. Press any key to resume scrolling.

The IBM *Personal Editor* utility also provides an excellent means for reviewing a listing file.

Create a Cross Reference

Using the .CRF file produced by the assembler, the CREF utility can produce a cross reference listing to aid in the debugging process. To create a cross reference, follow the instructions under "How to Start CREF" in Chapter 6 "Cross Reference Utility (CREF)" in this book.

When the utility is finished and DOS displays its prompt, you can display the .REF file contents by pressing the Ctrl-PrtSc keys (if you have a printer) and entering the DOS TYPE command. Press the Ctrl-PrtSc keys again to resume. The "DOS 2.0" (or later) "PRINT" command can be used to print the .LST or .REF files. The "DOS" "TYPE" command can be used to scroll the listing on the screen. It is not recommended to "COPY" a .LST file directly to the printer, because of tab problems.

Now that you have a printed listing of the source code (.LST) and the source code cross reference (.REF), these two files can be removed from the diskette by using the DOS ERASE command.

If you plan to keep these files on a diskette, the two files can be combined with the "DOS 2.0" (or later) "COPY" command. You can make the cross reference a part of the listing file, by:

```
COPY MODULE.LST+MODULE.REF
DEL  MODULE.REF
```

The IBM *Personal Editor* utility can also be used to review the cross reference listing.

Eliminate All Errors

If your source file contains any errors, reload an editor and revise the text.

Repeat the above steps (complete The Assembly Step then use an editor to revise the text) until each source module is error-free.

> **Note:** An error-free source module contains no assembler-recognizable syntax or code errors; however, it can contain logical errors that become apparent only to the user if the program does not produce the desired output.

The Linker Step

A complex application can logically be defined in a series of separate modules, keeping the complexity of any one module down to an acceptable level of understandability.

Once all modules of any particular application have been successfully assembled into object files, these modules can be processed by the Linker to form one composite load module.

Two LINK utilities are available to the user:

- DOS LINK (on the DOS diskette)
- Macro Assembler LINK (on the Macro Assembler diskette)

> **Note:** The Macro Assembler Linker is used only by DOS version 2.0. If your DOS version is less than 2.0, use the Linker supplied on your DOS diskette.

The Macro Assembler LINK utility supports:

- More external symbols
- Larger load modules
- Variable page sizes which can be defined in the various library formats

Use the Macro Assembler LINK utility (for DOS version 2.0), therefore, when linking object modules.

> **Note:** If you have only one object file, you must still use the linker to convert your file into a load module.

For detailed instructions on how to use the linker, see Chapter 8 "The Linker (LINK) Utility" in this book or consult your DOS book.

Viewing Linker Files

When linking is complete, control returns to DOS. Use the DOS DIR command to show the files produced by the linker. If you have created a .MAP file, press the Ctrl-PrtSc keys (if you have a printer) and enter the DOS TYPE command to see the memory map. Press the Ctrl-PrtSc keys again to return to normal display mode. After obtaining a printed copy of the memory map, you can erase the file.

Executing Your Program

You have now collected and linked all object modules to form one load module file *filename*.EXE.

If you have designed and encoded your program according to the rules for .COM files (see "Program Structure" in Chapter 3 "Memory and Segmentation" for details on .EXE and .COM files), the linker has returned an error message, "Warning: No STACK Segment"; expect and ignore this message. To execute your program, you must now use the DOS EXE2BIN command to convert your module from an .EXE format to the DOS Command format (.COM). (See "EXE2BIN" in the "Advanced Commands" section of the DOS book for details on how to use the EXE2BIN utility.)

> **Note:** Erase the .EXE file once it is converted to .COM format.

Your application will now run as a DOS external command.

To run your application, enter the application's filename (do not specify the extension). For example:

```
MODULE
```

DOS loads the application, decides where in storage to place it, resolves all references applying to that particular location, then gives control to the application. From here on, the machine is under control of the application program.

When the application run is complete, control returns to DOS.

You have now completed the assembly, linkage and execution of an application program.

Use DEBUG

To see what is occurring in your program, familiarize yourself with the DOS DEBUG utility. (See "The DEBUG Program" chapter in the DOS book for details on how to use DEBUG). DEBUG is a useful tool designed to aid the programmer in locating and correcting logical errors.

Notice these points when you are working with an assembler listing to debug and patch your programs:

1. Reversed Order of Bytes

When using DEBUG, be aware that single numeric values that occupy more than one byte are stored in memory in reverse of the order in which they are coded (the low-order byte is placed ahead of the high-order byte). For example:

In memory, at offset 0, a 2-byte integer whose value is decimal 10 is represented as:

```
0  [0A]    least significant byte first
1  [00]    most significant byte last
```

If you are reading a listing or using DEBUG, this reversal of byte positions is reflected on your output medium.

Some machine code is shown as individual bytes, separated by a single blank. Other machine code shows two bytes together, as a word immediate field or a word displacement. In memory, the contents of these two bytes are actually reversed from the order shown in the listing. For example:

```
0015  BA 001c R   MOV  DX,OFFSET MESSG
```

In memory, these three bytes are in this order:

```
BA  1C  00
```

Where the word offset is stored with its least significant byte first, followed by the most significant byte.

When patching in hex, reverse the order of the two bytes shown together as a word group.

2. IP or SEG Displacement

This example shows sample lines from a listing and the corresponding lines from a debug unassemble. Lines on listing:

```
0009  E8 0036 R      CALL  CLS
000C  BA 0000        MOV   DX,0
```

Debug lines read:

```
04AF:0009 E82A00     CALL 0036
04AF:000C BA0000     MOV  DX,0000
```

The 0036 indicates the relative location of the "CLS" symbol, defined as a PROC NEAR. A CALL or JMP instruction, however, has a word operand that is a displacement from the current value of the IP (instruction pointer). At execution time the IP has the address of the instruction following the CALL or JMP.

Compare the listing line with the actual code. The debug line shows the value in this word is not 0036, but 2A00. Once again, this word is in reversed order (least significant byte first). Also the "address" of 0036 has become a "displacement" from the next instruction at location 000C. If we add 000C and 002A (using the most significant byte first again), we get 0036, the value originally shown in the listing as being the generated value.

To patch an intrasegment CALL or JMP in hex, convert the address to a displacement from the next instruction, even though it is shown as an offset from the start of the segment.

3. Fields Modified by LINK or LOADER

Word displacements of any type instruction (where the word is followed by "R")may have been modified by the Linker and Loader to be some value other than that shown in the listing.

Because the Linker may have placed several segments of the same name together, the offset at run time is not the same as the offset was when assembling a portion of the final segment.

In general, be careful when patching in hex.

Professional Debug Facility

For extensive program debug tasks, consider the IBM Personal Computer *Professional Debug Facility.* This companion to the *Macro Assembler* provides:

- A multifunction, interactive, full-screen debug program.
- A program to display or change control and data areas of fixed disks or diskettes.
- A non-maskable interrupt hardware card to gain access to a "locked" system by pressing a button on the card.

This product is available through any authorized IBM product source.

Chapter 6. Cross Reference Utility (CREF)

Contents

Introduction **6-3**

Create a Symbol File **6-3**
    Before You Begin 6-4

How to Start CREF **6-4**
    Option 1 — Console Responses 6-4
    Option 2 — Command Line Responses 6-6
    CREF Termination 6-7
    CRF Format 6-8

Introduction

The Cross Reference Utility (CREF) is designed to aid in debugging source programs. CREF produces an alphabetic listing file of data, code, and segment-relative symbols. The file contains the line number where each symbol is defined (denoted by the # sign) and the line number(s) where they are referenced.

Create a Symbol File

To create a cross reference file, you must first create a symbol file (identified by the filename extension .CRF) using the assembler. The .CRF file is processed by CREF to produce the listing file *filename*.REF. (Note the difference in the filename extensions.)

During the assembly session, answer the fourth prompt, "Cross reference [NUL.CRF]:", from the assembler by entering a filename. (You must specify a filename to override the assembler default, "NUL", which does not produce a symbol file.)

The assembler produces a symbol file *filename*.CRF that contains a compact representation of the symbols' definitions and line references.

Before You Begin

This list provides information and suggestions for running the Cross Reference Utility.

Notes:

1. The name shown within the brackets is the *default* that the utility uses if you do not enter a filename of your own. It can be overridden.
2. Although the utility supplies a default filename extension if you do not specify one, the default extensions can be overridden by specifying the filename with the new extension.
3. The default diskette drive is the DOS default drive; it can be overridden by including the drive ID in front of the filename specification.

How to Start CREF

Option 1 — Console Responses

From the keyboard, type:

```
CREF
```

The Cross Reference Utility is loaded into memory and displays a series of two prompts, one at a time, to which you must enter the requested responses.

The two prompts and their corresponding response formats are:

PROMPT	RESPONSES
Cref filename [.CRF]:	[*d*:][*path*]*filename*[*.ext*]
List filename [*cross-ref*.REF]:	[*d*:][*path*][*filename*[*.ext*]]

PROMPT	DEFINITION
Cref filename	Is the name of the symbol file produced by the assembler.
List filename	Is the name you want to give to the cross reference listing file. "*cross-ref*" is the default name supplied by CREF if you press the Enter key in response.

Specify CON or USER to direct the listing to the operator's display, or COM1 or LINE to go to the RS-232 port, (see example in Option 2).

RESPONSES	DEFINITION
d:	For the .CRF file, the drive on which it resides. For the .REF file, the drive on which you want to place the new file. (If the drive is not specified, the default drive is used.)
path	For the .CRF file, the path used to locate the file's directory. For the .REF file, the path by which the new file is routed. (If the path is not specified, the default path is used.)
filename	Is the name of the file.
.ext	Is the filename extension of the file.

Option 2 — Command Line Responses

From the keyboard, type:

```
CREF cross-ref,list;
```

cross-ref	Is the name of the symbol file produced by the assembler.
list	Is the name you want to give to the cross reference listing file.

The CREF utility is loaded and immediately produces the listing file indicated in the command field.

If no list entry is given and no semicolon is used, CREF prompts for the file. The prompt displays its default, which can be accepted by pressing the Enter key, or overridden with an explicit filename (optionally including the drive ID), or a device name (CON or COM1). If an incomplete command is given and is terminated with a final semicolon, the unspecified file defaults without further prompting.

Examples:

CREF module

> The input cross reference file is module.CRF. A prompt is given, showing the default listing as module.REF. After a response is entered, the utility begins its operations.

CREF module;

> If the semicolon is added, no further prompts are displayed. The input cross reference file is module.CRF. The listing file is module.REF.

CREF module,CON

Because the complete list is specified, the presence or absence of the semicolon has no effect. The input cross reference file is module.CRF; the listing is sent directly to the operator's display. You can get hardcopy by echoing to your printer, if you have one.

CREF module,B:

This is the same as above, except the listing file is produced under the default filename of module.REF and allocated to drive B.

CREF Termination

When the Cross Reference Utility finishes running, control returns to DOS.

You may stop the execution of CREF prior to its normal exit by pressing the Ctrl and Break keys together.

CRF Format

You can use the CREF Utility for files other than those generated by the assembler. The format of the *module.CRF* file is:

First byte	Number of lines to be printed before advancing forms to the next top-of-page.
Second byte	Number of characters-per-printed line.
Third byte	Page length/width identifier for record type 07. **Note:** If the file does not begin with these 3 bytes, CREF defaults the page length to 58 and the page width to 80.
Records	The rest of the file is a series of records. Each record begins or ends with a byte that identifies the type of record as follows:

Records That Begin with a Record Type

Record Type Byte	Name	Subsequent Bytes
01	Reference symbol	Symbol name that is referenced (1-80 char.)
02	Define symbol	Symbol name that is defined (1-80 char.)
04	End of line	(none)
05	End of file	1AH
06	Title defined	Title text (1-80 char.)
07	Page length/width	1-byte length followed by 1-byte width.

Every line in a source program must produce at least one entry to the module.CRF file. For example, a COMMENT line that contains no references, produces a single hex 04 (End of line) record. This is how the CREF Utility maintains line numbers.

Chapter 7. The Library Manager (LIB)

Contents

Introduction **7-3**
  Terms 7-4
  Library Manager Functions 7-4

Using the Library Manager **7-5**
  Library Operators 7-7
  Entering Commands 7-8
    Examples 7-8
    Response File 7-10
    Cross Reference Listing 7-10

Introduction

The Library Manager utility allows you to build and edit a library of object modules. Once you have converted your error-free source file into an object module (using either the assembler or a compiler), the module can be stored in your library and recalled for use in a future program. When a module is needed from your library, you simply supply the linker with the library name. You need not remember the name of the individual module. The Library Manager thus frees you from several file maintenance tasks.

The use of a library promotes the development of modular programming practices:

- The modular approach to program construction is the foundation of structured programming.
- A well designed, general purpose module can be re–used many times.
- The creation and maintenance of a library of general purpose modules can significantly reduce program development time.

Terms

The following terms are used throughout this chapter:

Object Module	An object module is an assembled unit of code ready to be linked.
Object File	An object file is a DOS file containing *exactly one* object module.
Library	A library file is a collection of object modules. This is also called a library.
Library Manager	A library manager is a utility program for the construction and maintenance of object module library files.

Library Manager Functions

The Library Manager can perform these functions:

- Add an object file
- Delete an object module
- Extract an object module
- Replace an object module
- Add other libraries
- List the contents of a library
- Change the internal page size used to store the contents of the library.

Using the Library Manager

All commands to the Library Manager can be entered on the command line at the DOS prompt.

```
LIB library file[page size][operations],[list file],[new library]...[;]
```

PROMPT	**DEFINITION**
library file	Is the DOS file name of a library file.
page size	Is an option to change the internal page size used to store members of the library. Individual modules are aligned to begin at the beginning of a new page. If a module does not occupy the number of bytes the exact multiple of the page size, then the last block is padded with as many null bytes required to fill the page. Large page sizes reduce link time. Small page sizes reduce library size. **Note:** The page size defaults to 512 bytes. Version 2.2 of the linker is included with this version of the IBM Macro Assembler. Previous versions of the linker cannot recognize page sizes other than 512. The format of the page size is: `/pagesize:N` where N equals 2**n and $4 \leq n \leq 15$.

Example entries:

```
/pagesize:16    (smallest page size allowed)
/pagesize:32
/pagesize:64
/pagesize:128
.
.
.
/pagesize:16348
/pagesize:32768  (largest page size allowed)
```

operations	Is a list of the actions you want the Library Manager to perform. This list contains an operator plus the name of the file you are adding, deleting, or replacing. The default is an empty list; no changes occur. See Library Operators, later in this section.
list file	Is a DOS file name where a cross–reference listing of the library contents is placed. No default extensions are used. The default is no list file generated.
new library	Is the DOS file name of a library file to be created with the changes specified by the operations or the page size or both. If you use the default, the original file is renamed to have the extension ".BAK", and the new library is created using the original file name.
;	The semicolon at the end causes all remaining fields to assume their default values. You can end the command anyplace after the *library name* with a semicolon, forcing all the fields following the semicolon to assume their default values.

Note: The *library name* field is always required.

Library Operators

The Library Manager recognizes the following operators:

OP	DEFINITION
+	Adds the contents of an object file or a library file.
-	Deletes an object module
*	Copies an object module into a file whose name is the specified module name plus the extension ".OBJ".

These basic operators can be combined to do more complex operations. For example:

OP	DEFINITION
-+	Replaces an object module with the contents of the object file of the same name (plus the extension .OBJ)
-*	Deletes and extracts an object module.

You can do as many operations at once as you wish.

You are not limited by the length of the line on the screen. If you wish to continue a command on the next line, end the current line with the ampersand character (&) immediately followed by a carriage return.

Entering Commands

There are several ways to issue a command to the Library Manager:

- You can type all the parameters at once, then enter the complete command. Any empty field assumes the default value.
- You can press the Enter key at any point after the LIB command and you are prompted by Library Manager for each remaining part. Any empty field assumes the default value.
- You can end the command anyplace after the *library name* with a semicolon, forcing all the fields following the semicolon to assume their default values.

  Note: The *library name* field is always required.
- You can issue commands from a *response file*.

Examples

To add file SCREEN.OBJ to the library PASCAL.LIB without producing a cross reference, you would type:

```
   LIB PASCAL.LIB + SCREEN.OBJ;
or
   LIB PASCAL + SCREEN;
```

To copy the module SCREEN from the library PASCAL.LIB into the file SCREEN.OBJ, you would enter:

```
LIB PASCAL * SCREEN;
```

To extract and place the module SCREEN into the file SCREEN.OBJ and delete it from PASCAL.LIB, you would enter:

```
LIB PASCAL -* SCREEN;
```

Note: These two examples are equivalent because extensions are optional, and default to .OBJ for operations and .LIB for library file, if omitted. If you are using a library file that is in the operations list, you must specify the .LIB extension. The semicolon at the end causes all remaining fields to assume their default values, and no prompts are issued.

To delete SCREEN from PASCAL.LIB, you would type:

```
LIB PASCAL - SCREEN;
```

To replace SCREEN in the library with a newer version, you would type:

```
   LIB PASCAL -+ SCREEN;
or
   LIB PASCAL - SCREEN + SCREEN.OBJ;
or
   LIB PASCAL - SCREEN + SCREEN;
```

If you want to make the same change as above, but want to put the changes in a library called PASNEW, then any of these entries works:

```
   LIB PASCAL -+ SCREEN,,PASNEW
or
   LIB PASCAL - SCREEN + SCREEN,,PASNEW
or
   LIB PASCAL + SCREEN - SCREEN,,PASNEW
```

Note: The last two examples are equivalent because of the way operations are ordered in the Library Manager. Deletions and copies are carried out first and in the order in which specified object modules occur in the library. Additions are then carried out in the order that you specify.

Response File

You can issue commands from a response file. The following example causes the Library Manager to read its responses from a file named RESP.TXT.

```
LIB @RESP.TXT
```

Note: The name RESP.TXT was chosen merely as an example. Your file can have any valid DOS filename.

Cross Reference Listing

If a file name is inserted between the commas in the previous example, a cross-reference file with that name would be created on the disk. The next example does all the operations of the previous example and also creates a cross-reference of the contents of the library in a disk file named PASLIST.

```
LIB PASCAL + SCREEN - SCREEN,PASLIST,PASNEW
```

The cross-reference listing can be displayed on the screen using the DOS TYPE command, and can be printed using the DOS COPY or DOS PRINT command:

```
TYPE filespec

COPY filespec PRN

PRINT filespec
```

The Ctrl-PrtSc and Shift-PrtSc functions can also be used to print the listing file, along with the DOS TYPE command above.

Chapter 8. The Linker (LINK) Utility

Contents

Introduction **8-3**

Files **8-4**
  Input Files 8-4
  Output Files 8-4
  VM.TMP (Temporary File) 8-5

Definitions **8-5**
  Segment 8-6
  Group 8-6
  Class 8-7

Command Prompts **8-7**

Detailed Descriptions of the Command Prompts . . **8-9**
  Object Modules [.OBJ]: 8-9
  Run File [filename.EXE]: 8-10
  List File [NUL.MAP]: 8-11
  Libraries [.LIB]: 8-12
  Linker Parameters 8-14
    /DS ALLOCATION 8-14
    /HIGH 8-14
    /LINE 8-15
    /MAP 8-15
    /NODEFAULT LIBRARY SEARCH . 8-15
    /NOGROUP ASSOCIATION 8-15
    /PAUSE 8-16
    /STACK:size 8-16

Starting the linker Program **8-17**
  Before You Begin 8-17
  Option 1 — Console Responses 8-17
  Option 2 — Command Line Responses . . . 8-18
  Option 3 — Automatic Responses 8-20

Example linker Session **8-22**
Determining the Absolute Address of a Segment 8-25

Messages **8-26**

Introduction

A new, more powerful linker is now included with the Macro Assembler. This version of LINK can only be used with DOS version 2.0 or later. This chapter describes how to use the IBM Personal Computer LINK utility. If you are using this linker for the first time, we recommend that you read all of this chapter before you use LINK.

The Linker (LINK) Utility:

- Combines separately produced object modules
- Searches library files for definitions of unresolved external references
- Resolves external cross-references
- Produces a printable listing that shows the resolution of external references and error messages
- Produces a relocatable load module.

Files

The linker processes the following input, output, and temporary files:

Input Files

Type	Default .ext	Override .ext	Produced by
Object	.OBJ	Yes	Assembler or Compiler[1]
Library	.LIB	Yes	Compiler
Automatic Response	(None)	N/A*	User

`N/A* - Not applicable.`

Output Files

Type	Default .ext	Override .ext	Used by
Listing	.MAP	Yes	User
Run	.EXE	No	Relocatable loader (COMMAND.COM)

---

[1] One of the optional compiler packages available for use with the IBM Personal Computer *Disk Operating System (DOS).*

VM.TMP (Temporary File)

LINK uses as much available memory as necessary to hold the data that defines the load module being created. If the module is too large to be processed in available memory, the linker may need additional memory space. If this happens, a temporary file called VM.TMP is created on the DOS default drive.

When the overflow to the VM.TMP file has begun, the linker displays the following message:

```
VM.TMP has been created
Do not change diskette in drive x
```

If the VM.TMP file has been created on diskette, do not remove the diskette until LINK ends. When LINK ends, it erases the VM.TMP file.

If the DOS default drive already has a file named VM.TMP, it is deleted by LINK and a new file is allocated; the contents of the previous file are destroyed. Avoid using VM.TMP, therefore, as one of your own filenames.

Definitions

Segment, *group*, and *class* are attributes discussed in Chapter 2 “Assembly Language Format”. These terms describe the underlying function of LINK; understand these terms to gain a basic understanding of the way LINK works.

Segment

A *segment* is a contiguous area of memory up to 64K bytes long. A segment can be located anywhere in memory on a *paragraph* (16-byte) boundary. Each of the four segment registers defines a segment. The segments can overlap. Each 16-bit address is an offset from the beginning of a segment. The contents of a segment are addressed by a segment register/offset pair (see Chapter 3 "Memory and Segmentation" for details).

The contents of various portions of the segment are determined when the assembler generates object code (machine language). Neither the size nor location of a segment is necessarily fixed by the assembler because this portion of the segment can be combined at link time with other portions forming a single segment.

A program's ultimate location in memory is determined at load time by the relocation loader in COMMAND.COM, based on whether you specified the /HIGH parameter. (See the /HIGH parameter in the "Linker Parameters" section of this chapter.)

Group

A *group* is a collection of segments that fit together within a 64K-byte segment of memory. The segments are named to the group by the assembler. A program can consist of one or more groups.

The group is used for addressing segments in memory. The various portions of segments within the group are addressed by a segment base pointer plus an offset.

> **Note:** The linker does not check that the object modules of a group meet the 64K-byte constraint.

Class

A *class* is a collection of segments. The naming of segments to a class affects the order and relative placement of segments in memory. The class name is specified by the assembler. All portions assigned to the same class name are loaded into memory contiguously.

Segments are ordered within a class in the order that the linker encounters the segments in the object files. One class precedes another in memory only if a segment for the first class precedes all segments for the second class in the input to LINK. Classes are not restricted in size.

Command Prompts

After you start the linker session, you receive a series of four prompts. You can respond to these prompts from the keyboard, respond to these prompts on the command line, or you can use a special diskette file called an *automatic response file* to respond to the prompts.

LINK prompts you for the names of the object, run, list, and library files. When the session is finished, LINK returns to DOS and the DOS prompt is displayed. If linking is unsuccessful, LINK displays a message.

The prompts are described in order of their appearance on the screen. Defaults are shown in square brackets ([ ]) after the prompt. In the response column of the table, square brackets show optional entries. **Object Modules** is the only prompt that requires a response from you.

PROMPT	RESPONSES
Object Modules [.OBJ]:	[*d*:][path]*filename*[.*ext*] [+[*d*:][path]*filename*[.*ext*]]...
Run File [filename.EXE]:	[*d*:][path][*filename*[.*ext*]]
List File [NUL.MAP]:	[*d*:][path][*filename*[.*ext*]]
Libraries [.LIB]:	[*d*:][[path]*filename*[.*ext*]] [+[*d*:][[path]*filename*[.*ext*]]]...

Notes:

1. The name shown within the brackets is the *default* filename extension that the linker uses if you do not choose a filename extension of your own.

2. For dual-drive systems, the default diskette drive is the DOS default drive; it can be overridden by including the drive ID as part of the filespec. If you enter a filename without specifying the drive, the default drive is assumed.

3. For DOS version 2.0, the default path is the current directory. Other sub-directories can be referenced by indicating the path name before the filename. If you enter a filename without specifying the path, the default path is assumed. The libraries prompt is an exception—the linker looks for the libraries on the default drive; if not found, it looks on the drive specified by the compiler.

4. Parameters (*/parms*) are never prompted; they can be added to the end of the command line (before the semicolon) or to any file specification given in response to a prompt.

5. You can end the linker session before its normal end by pressing Ctrl-Break.

Detailed Descriptions of the Command Prompts

The following detailed descriptions contain information about the responses that you can enter to the prompts.

Object Modules [.OBJ]:

Enter one or more file locations for the object modules to be linked. Multiple file locations must be separated by single plus (+) signs or blanks. If the extension is omitted from any filename, LINK assumes the filename extension **.OBJ**. If an object module has a different filename extension, the extension must be specified. Object filenames cannot begin with the @ symbol (@ is reserved for using an automatic response file).

LINK loads segments into classes in the order encountered.

If you specify an object module on a diskette drive, but LINK cannot find the file, it displays the following prompt:

```
Cannot find file object module
change diskette <hit ENTER>
```

If you specify an object module on a non-removable media (like a fixed disk), the linker session ends with the following message:

```
Cannot find file object module
```

Insert the diskette containing the requested module. This permits **.OBJ** files from several diskettes to be included. On a single-drive system, diskette exchanging can be done safely *only* if VM.TMP has *not* been opened. As explained earlier in the "VM.TMP (Temporary File)" section, a message shows if VM.TMP has been opened.

IMPORTANT: If a VM.TMP file has been opened on a diskette, do *not* remove the diskette containing the VM.TMP file.

If a VM.TMP file is opened after you specify an object module on the disk where VM.TMP resides and LINK cannot find it, the linker session ends with the message:

```
Cannot find file object module
```

Run File [filename.EXE]:

The file specification you enter is created to store the run (executable) file that results from the LINK session. All run files receive the filename extension **.EXE**, even if you specify another extension.

The default filename for the run file prompt is the first filename specified on the object module prompt.

You can specify just a drive letter, or a path on the run file prompt. Either specification changes the place where the run file *filename*.EXE is placed.

List File [NUL.MAP]:

The linker list file is sometimes called the linker *map*.

The list file is not created unless you specifically request it. You can request it by overriding the default with a drive letter, path, or *filename*[*.ext*]. If you do not include a filename extension, the default extension **.MAP** is used. If you do not enter anything, the DOS reserved filename **NUL** specifies that no list file will be created.

The list file contains an entry for each segment in the input (object) modules. Each entry also shows the offset (addressing) in the run file.

You can specify just a drive letter or a path on the list-file prompt. This changes the place where the list file is placed.

> **Note:** If the list file is allocated to a file on diskette, that diskette must not be removed until the LINK has ended.

If you specify an object module on the same diskette drive as the diskette drive to which the list file is allocated, and LINK cannot find the object module, the linker session ends with the message:

```
Cannot find file object module
```

To avoid generating the list file on a diskette, you can specify the display or printer as the list file device. For example:

```
   List File [NUL.MAP]: CON
```

If you direct the output to your display, you can also print a copy of the output by pressing Ctrl-PrtSc.

Libraries [.LIB]:

You can either list the file locations of any user libraries, or press the Enter key. If the object files are generated by the assembler only, pressing Enter tells LINK that no libraries are being used (no library search is requested). If the object files are generated by a compiler, pressing the Enter key causes LINK to look for the default library (provided as part of the Compiler package) on the default drive. If the library is not on the default drive, LINK looks for the library on drive A. If the library is not on drive A, LINK prompts for the filespec.

If you answer the library prompt, you specify a list of drive letters and *[path]filename.ext* separated by plus signs (+) or spaces. You can enter from one to eight library file locations. Specifying a drive letter tells linker to look on that drive instead of the Compiler package-supplied drive for all later libraries on the library prompt. The automatically searched library file specifications are conceptually placed at the end of the response to the library prompt.

LINK searches the library files in the order which they are listed to resolve external references. When LINK finds the module that defines the external symbol, the module is processed as another object module.

If two or more libraries have the same filename, regardless of the location, only the first library in the list is searched.

When LINK cannot find a library file, it displays a message like this:

```
  Cannot find library A:library file
Enter new drive letter:
```

One of the following responses is required:

- The drive ID on which the indicated library is located must be entered.
- The drive ID and PATH on which the indicated library is located must be entered.

  Note: PATH must end with a backslash (\) or a new library name.

The following library prompt responses can be used:

```
Libraries [.LIB]: B:
```

Look for compiler.LIB on drive B.

```
Libraries [.LIB]: B:USERLIB
```

Look for USERLIB.LIB on drive B and compiler.LIB on drive A.

```
Libraries [.LIB]: A:LIB1+LIB2+B:LIB3+A:
```

Look for LIB1.LIB and LIB2.LIB on drive A, LIB3.LIB on drive B, and compiler.LIB on drive A.

Linker Parameters

At the end of any of the four linker prompts, you can specify one or more parameters that instruct the linker to do something differently. Only the / and first letter of any parameter are required, for all but two parameters; they are /NOD, and /NOG.

/DS ALLOCATION

The /DS ALLOCATION (/D) parameter directs LINK to load all data defined in DGROUP at the *high end* of the group.

If the /DS ALLOCATION parameter is not specified, LINK loads all data defined in DGROUP at the *low end* of the group, beginning at offset 0. The only storage thus referenced by the data space pointer is that storage specifically defined as residing in the group.

All other segments of any type in any GROUP other than DGROUP are loaded at the low end of their respective groups, as if the /DS ALLOCATION parameter were not specified.

/HIGH

The /HIGH (/H) parameter causes the loader to place the run image as high as possible in storage without overlaying the transient portion of COMMAND.COM, which occupies the highest area of storage. If you do not specify the /HIGH parameter, the linker directs the loader to place the run file as low in memory as possible.

The /HIGH parameter can be used with the /DS ALLOCATION parameter.

/LINE

For certain IBM Personal Computer language processors, the /LINE (/L) parameter directs LINK to include the line numbers and addresses of the source statements of input modules in the list file.

/MAP

The /MAP (/M) parameter directs LINK to list all public (global) symbols that are defined in the input modules. For each symbol, LINK lists its value and segment-offset location in the run file. The symbols are listed at the end of the list file.

/NODEFAULT LIBRARY SEARCH

The /NODEFAULT LIBRARY SEARCH (/NOD) parameter directs LINK not to search for default libraries. The default libraries are those requested by BASIC, Fortran, and Pascal objects for example.

/NOGROUP ASSOCIATION

The /NOGROUP ASSOCIATION (/NOG) parameter directs LINK to fix-up external long addresses off the segment base even if the symbol was defined in a segment that was part of a group.

/PAUSE

The /PAUSE (/P) parameter tells LINK to display this message:

```
About to generate .EXE file
Change disks <hit ENTER>
```

This message allows you to insert the diskette containing the run file.

> **Note:** The /P parameter is suppressed if the list file is on the same drive as the .EXE (run) file is destined for.

/STACK:size

The *size* entry is any positive decimal value up to 65,536 bytes. This value is used to override the size of the stack that is specified in the source file's stack segment. (If the stack is produced by a compiler, it can also be overridden.) If you specify a value greater than 0 but less than 512, the value 512 is used.

If you do not specify /STACK (/S), the original stack size is used.

If the size of the stack is too small, the results of executing the resulting load module are unpredictable.

Unless you have designed your application according to specifications for .COM files, at least one input (object) module must contain a stack allocation statement. The DOS loader automatically sets up the stack for .COM designed source files. For .EXE designed files, the source must contain a SEGMENT command that has the combine type of STACK (see "Program Structure" in Chapter 3 "Memory and Segmentation"). If a stack allocation statement is not provided, LINK returns a **Warning: No Stack statement** message.

Starting the linker Program

Before You Begin

- Make sure the files you will be using for linking are on the appropriate disks.
- Make sure you have enough free space on your disks to contain your files and any generated data.

You can start the linker program by using one of three options:

Option 1 — Console Responses

From your keyboard, enter:

```
LINK
```

The linker is loaded into memory and displays a series of four prompts, one at a time, to which you must enter the requested responses. (Detailed descriptions of the responses that you can make to the prompts are discussed in this chapter.)

If you enter a wrong response, such as an incorrectly spelled filename, you must press Ctrl-Break to exit LINK, then restart LINK. If the response in error has been typed but you haven't pressed Enter yet, you can delete the wrong characters (on that line only).

An example of a linker session using the console response option is provided in this chapter in the section called "Example linker Session."

When you have entered the last filename, the linker begins to run. If the linker finds any errors, it displays the errors on the screen and in the listing file.

Option 2 — Command Line Responses

From your keyboard, enter:

```
LINK objlist,runfile,mapfile,liblist[parm]...[;]
```

PROMPT	**DEFINITION**
objlist	Is a list of object modules separated by spaces or plus signs (+).
runfile	Is the name you want to give the run file.
mapfile	Is the name you want to give the linker map.
liblist	Is a list of the libraries to be used, separated by plus signs (+) or spaces.
parm	Is an optional linker parameter. Each parameter must begin with a slash (/).
;	Is a signal to end the command and use the defaults, if an incomplete list is entered.

The linker is loaded and immediately carries out the tasks indicated by the command line.

When you use this command line, the prompts described in Option 1 are not displayed if you specified an entry for all four files or if the command line ends with a semicolon.

If an incomplete list is given and no semicolon is used, the linker prompts for the remaining unspecified files.

Each prompt displays its default, which can be accepted by pressing the Enter key, or overridden with an explicit filename or device name. However, if an incomplete list is given and the command line is terminated with a final semicolon, the unspecified files default without further prompting. The *parms* are never prompted for, but can be added to the end of the command line or to any file specification given in response to a prompt.

Certain variations of this command line are permitted.

Examples:

LINK module

The object module is MODULE.OBJ. A prompt is given, showing the default of MODULE.EXE. After entering the response, a prompt shows the default of NUL.MAP. After giving the response, a prompt displays the default extension of .LIB.

LINK module;

If you add the semicolon, no further prompts are displayed. The object module of MODULE.OBJ is linked, the run file is put into MODULE.EXE, and no list file is produced.

LINK module,,;

This is similar to the preceding example, except the list file is produced in MODULE.MAP.

LINK module,,

Using the same example, but without the semicolon, MODULE.OBJ is linked, and the run file is produced in MODULE.EXE, but a prompt is given with the default of MODULE.MAP.

LINK module,,NUL;

No list file is produced. The run file is in MODULE.EXE. No further prompts are displayed.

Option 3 — Automatic Responses

It is often convenient to save responses to the linker for use at a later time. This is especially useful when long lists of object modules need to be specified.

Before using this option, you must create the automatic response file. It contains several lines of text, each of which is the response to a linker prompt. These responses must be in the same order as the linker prompts that were discussed earlier in this chapter. If desired, a long response to the object module or libraries prompt can be contained on several lines by using a plus sign (+) to continue the same response on the next line.

To specify an automatic response file, enter a file specification preceded by an @ symbol in place of a prompt response or part of a prompt response. The prompt is answered by the contents of the diskette file. The file specification cannot be a reserved DOS filename.

From your keyboard, enter:

```
LINK @[d:][path]filename1[ext]
```

The filename extension is optional and can be any name. There is no default extension.

This option permits the command that starts LINK to be entered from the keyboard or within a batch file without requiring any response from you.

Example

```
Automatic Response File - RESP1

   MODA+MODB+MODC+
   MODD+MODE+MODF+

Automatic Response File - RESP2

   runfile/p
   printout

Command line

   LINK @RESP1+mymod,@RESP2;
```

Notes:

1. The plus sign at the end of the first line in RESP1 causes the listed modules on both lines to be read as input object modules. After reading RESP1, the linker returns to the command line and sees **+mymod**, so it includes MYMOD.OBJ in the first list of object modules as well.

2. Each of the above lines ends when you press the Enter key.

Example linker Session

This example shows you the type of information that is displayed during a linker session.

Once you enter:

```
b:link
```

in response to the DOS prompt, the system responds with these messages and prompts, which we answer as shown:

```
IBM Personal Computer Linker
Version 2.20 (C)Copyright IBM Corp. 1981, 1982, 1983, 1984

Object Modules [.OBJ]: example
Run File [EXAMPLE.EXE]: /map
List File [NUL.MAP]: prn/line
Libraries [.LIB]:
```

Notes:

1. Specifying **/map**, gives both alphabetic and chronological listings of public symbols.
2. Responding **prn** to the list file prompt, sends our output to the printer.
3. By specifying the /LINE parameter, LINK gives us a listing of all line numbers for all modules. (The /LINE parameter can generate a large amount of output.)
4. By just pressing Enter in response to the libraries prompt, an automatic library search is performed.

Once LINK locates all libraries, the linker map displays a list of segments in the relative order of their appearance within the load module. The list looks like this:

```
    Start  Stop   Length  Name           Class

    00000H 00028H 0029H   MAINQQ         CODE
    00030H 000F6H 00C7H   ENTXQQ         CODE
    00100H 00100H 0000H   INIXQQ         CODE
    00100H 038D3H 37D4H   FILVQQ_CODE    CODE
    038D4H 04921H 104EH   FILUQQ_CODE    CODE
.
.
.
    074A0H 074A0H 0000H   HEAP           MEMORY
    074A0H 074A0H 0000H   MEMORY         MEMORY
    074A0H 0759FH 0100H   STACK          STACK
    075A0H 07925H 0386H   DATA           DATA
    07930H 082A9H 097AH   CONST          CONST
```

The information in the **Start** and **Stop** columns shows a 20-bit hex address of each segment relative to location zero. Location zero is the beginning of the load module. The addresses displayed are not the absolute addresses of where these segments are loaded. To find the absolute address of where a segment is actually loaded, you must determine where the segment (listed as being at relative zero) is actually loaded; then add the absolute address to the relative address shown in the linker map. The procedure you use to determine where relative zero is actually located is discussed in this chapter, in the next section called "Determining the Absolute Address of a Segment."

Now, because we specified the /MAP parameter, the public symbols are displayed by name and by value. For example:

```
   Address          Publics by Name

   0492:0003H       ABSNQQ
   06CD:029FH       ABSRQQ
   0492:00A3H       ADDNQQ
   06CD:0087H       ADDRQQ
   0602:000FH       ALLHQQ
.
.
.
   0010:1BCEH       WT4VQQ
   0010:1D7EH       WTFVQQ
   0010:1887H       WTIVQQ
   0010:19E2H       WTNVQQ
   0010:11B2H       WTRVQQ

   Address          Publics by Value

   0000:0001H       MAIN
   0000:0010H       ENTGQQ
   0000:0010H       MAINQQ
   0003:0000H       BEGXQQ
   0003:0095H       ENDXQQ
.
.
.
   F82B:F31CH       CRCXQQ
   F82B:F31EH       CRDXQQ
   F82B:F322H       CESXQQ
   F82B:F5B8H       FNSUQQ
   F82B:F5E0H       OUTUQQ
```

The addresses of the public symbols are in the *segment:offset* format, showing the location relative to zero as the beginning of the load module. In some cases, an entry can look like this:

```
   F8CC:EBE2H
```

This entry appears to be the address of a load module that is almost 1 megabyte in size. Actually, the area being referenced is relative to a segment base that is pointing to a segment below the relative zero beginning of the load module. This condition produces a pointer that has effectively gone negative. The next memory map example illustrates this point.

When LINK has completed, this message displays:

```
Program entry point at 0003:0000
```

Determining the Absolute Address of a Segment

The linker map displays a list of segments in the relative order of their appearance within the load module. The information displayed shows a 20-bit hex address of each segment relative to location zero. The addresses displayed are not the absolute addresses of where these segments are located. To determine where relative zero is actually located, use DEBUG. DEBUG is described in detail in "The DEBUG Program" chapter of the DOS book.

Using DEBUG,

1. Load the application. Note the segment value in CS and the offset within that segment to the entry point as shown in IP. The last line of the linker map also describes this entry point, but uses relative values, not the absolute values shown by CS and IP.

2. Subtract the relative entry as shown at the end of the map listing from the CS:IP value. For example, let's say CS is at 05DC and IP is at zero.

   The linker map shows the entry point at 0100:0000. (0100 is a segment ID or paragraph number; 0000 is the offset into that segment.)

   In this example, relative zero is located at 04DC:0000, which is 04DC0 absolute.

If a program is loaded low, the relative zero location is located at the end of the Program Segment Prefix, in the location DS plus 100H.

Messages

All messages, except for the warning messages, cause the LINK session to end. Therefore, after you find and correct a problem, you must rerun LINK.

Messages appear both in the list file and on the display unless you direct the list file to CON, in which case the display messages are suppressed.

> **Note:** The linker does not accept segments that are exactly 64K long. If detected, an error message "Big bit not implemented" will issue.

See Appendix A for a complete listing of all linker messages.

Chapter 9. 80286-Based IBM Personal Computers

Contents

Introduction **9-3**
Modes of Operation 9-3
Base Architecture 9-4
Memory Access Differences 9-5

80287 Math Co-processor **9-6**

Introduction

The IBM Personal Computer Macro Assembler supports 80286-based IBM Personal Computers running in Real Address Mode. The IBM Personal Computer Macro Assembler also supports 80286-based IBM Personal Computers with the 80287 math co-processor feature running in Real Address Mode.

Modes of Operation

The 80286 and 80287 is capable of running either of two modes of operation. The two modes of operation are:

- Real Address Mode
- Protected Virtual Address Mode

Initially, upon system reset, the 80286 and 80287 starts up in Real Address Mode. In this mode of operation, all memory addressing is performed in terms of real physical addresses. In effect, the architecture of the 80286 and the 80287 in this mode is identical to that of the 8088 and the 8087. In Real Address Mode, the base architecture of the 80286 and 80287 is available. All future reference to the 80286 and the 80287 within the IBM Personal Computer Macro Assembler assumes Real Address Mode of operation.

Base Architecture

The base architecture of the 80286 has many features in common with the architecture of the 8088, such as byte addressable memory, I/O interfacing hardware, interrupt vectoring, and support for a math co-processor.

The 8088 and 80286-based IBM Personal Computers have a common set of addressing modes and basic instructions. The 80286 base architecture also includes a number of new instructions which add to the versatility of the computer. For example:

- POPA pops all general registers.
- PUSH immediate, pushes immediate data.
- PUSHA pushes all general registers.
- INS inputs a string.
- OUTS outputs a string.

Some of the new instructions give hardware support for procedure invocations, parameter passing, and array bounds checking.

> **Note:** See the IBM Personal Computer *Macro Assembler Reference* book for detailed information on 8087, 8088, 80286 and 80287 instructions.

Memory Access Differences

The 8088-based IBM Personal Computer always accesses memory in bytes. Word operands are accessed in two bus cycles regardless of their alignment. Although alignment of word operands does not affect the performance of the 8088, locating 16-bit data on even addresses insures maximum throughput if the program is ever transferred to an 80286-based IBM Personal Computer.

The 80286-based IBM Personal Computer can access either 8 or 16 bits of memory at a time. If an instruction refers to a word variable, and that variable is located at an even address, the 80286 accesses the complete word in one bus cycle. If the word is located at an odd address, it is accessed one byte at a time in two consecutive bus cycles. Because of instruction prefetching and queueing, there is no requirement for instructions to be aligned on word boundaries and no performance loss if they are not.

To maximize throughput in 80286-based IBM Personal Computers 16-bit data should be stored at even addresses (word aligned).

> **Note:** See the EVEN pseudo-op in Chapter 2 of the *Macro Assembler Reference* book for detail.

This is particularly true of stacks. Unaligned stacks can slow a system's response to interrupts. The alignment of stacks default to a Paragraph (16-byte) boundary. Unaligned stacks only occur when specified as a byte boundary alignment type. Except for the performance penalty, alignment is transparent to software.

80287 Math Co-processor

The 80286-based IBM Personal Computers support the 80287 math co-processor feature. In Real Address Mode, the architecture of a 80287 is similar to a 8087 math co-processor.

> **Note:** See the following chapter for detail on 8087 architecture.

The 80287 math co-processor feature uses the same instruction set as the 8087 math co-processor feature on the 8088-based IBM Personal Computer.

Chapter 10. IBM Personal Computer 8087 Math Co-Processor

Contents

Introduction **10-3**

8087 Architecture **10-5**
  Environment Data 10-5
    Environment Data Format 10-6
  Save State 10-6
    Save State Format 10-7
  Exception Pointers 10-8
    Exception Pointer Format 10-8
  Register Stack 10-9
    Register Stack 10-9
  Status Word 10-11
    Status Word (high-byte) 10-11
    Status Word (low-byte) 10-13
  Control Word 10-15
    Control Word (high-byte) 10-15
    Control Word (low-byte) 10-16
  Exceptions 10-18
    User-written Exception Handler 10-19
  Tag Word 10-21
    Tag Word (high-byte) 10-21
    Tag Word (low-byte) 10-21

Data Types **10-22**
  Selecting Data Types 10-23

Data Formats **10-24**
  Binary Integers 10-24
    Binary Integer Formats 10-25
  Decimal Integers 10-26
    Packed Decimal Format 10-26
  Real Numbers 10-27
    Real Number Formats 10-29

Temporary Real Format 10-30
Special Values 10-31

Data Format Requirements 10-33
8087 Real Number Formats 10-33
Microsoft Binary Real Format 10-35
Data Format Conversion 10-37
Data Conversion Routines 10-38
Using The ASCII - 8087 Conversion Routines 10-39
Using The Microsoft - 8087 Conversion Routines 10-43

Processor Control and Monitoring 10-45
Co-processing 10-45
Initialization 10-45
Interrupts and Waits 10-47
Interrupt Requests 10-47
Endless Wait 10-49

Introduction

The IBM Personal Computer 8087 Math Co-Processor option #1501002 allows the IBM Personal Computer to do high speed-arithmetic, logarithmic functions, and trigonometric operations with extreme accuracy.

The 8087 co-processor works in parallel with the 8088 processor. The parallel operation decreases operation time by allowing the 8087 to do mathematical calculations while the 8088 continues to do other functions.

With appropriate software, this option can significantly improve the speed and precision of doing these operations.

The Macro Assembler can assemble source programs for the 8087 math co-processor. At assembly time, it is necessary to specify the /R parameter, or have the .8087 pseudo-op in the source, if the 8087 math co-processor is is to be resident in the machine that will be running the application. If 8087 emulation (available in the FORTRAN and PASCAL libraries) is used, specify the /E parameter to generate external linkage. See Chapter 5 "Macro Assembler" in this book for details on assembly and parameter selection for 8087 instructions, and the FLINS.ASM file on the Macro Assembler diskette for examples of 8087 instructions.

When generating floating-point variables, the assembler uses the 8087 if installed, otherwise it will use its own (included) emulation routines for this operation. Therefore, for large assemblies of floating-point tables, the performance of the assembler will be faster, if the 8087 is present.

The instructions mnemonics for the 8087 are preceded by the letter "F", for example FADD, or FSUB. The first 5 bits of every instruction opcode for the 8087 are the same (11011 binary). These instructions are called "ESCAPE" instructions. When the 8088 and the 8087 see this instruction opcode, the 8088 calculates the address of any variables in memory, while the 8087 checks the instruction. The 8087 then takes the memory address from the 8088 if the instruction uses a memory operand.

To access locations in memory, the 8087 takes the local bus from the 8088 when the 8088 finishes its current instruction. When the 8087 is finished with the memory transfer, it returns the local bus to the 8088.

The 8087's busy signal informs the 8088 that it is executing an instruction, and the 8088's wait instruction tests this signal to find out when the 8087 is ready to execute following instructions.

The 8087 can be viewed as an architectural extension of the 8088 processor. It extends the register resources and instruction set of the IBM Personal Computer.

8087 Architecture

The 8087 is internally divided into two processing elements, the control unit and the numeric execution unit. The numeric execution unit executes all numeric instructions, while the control unit fetches instructions, reads and writes memory operands, and executes the 8088 control class of instructions.

The numeric execution unit executes all instructions that involve the register stack. These include arithmetic, comparison, transcendental, constant, and data transfer instructions.

The data path in the numeric execution unit is 80 bits wide (64 fraction bits, 15 exponent bits, and a sign bit), which allows internal operand transfers to be carried out at very high speeds.

Environment Data

The environment data can be stored in memory for examination, or while another routine takes control, with the FSTENV, or FNSTENV instruction. FLDENV reloads the 8087 environment data from memory. The environment data consists of seven words indicating the current state of the 8087.

- Control word
- Status word
- Tag word
- Instruction pointer (two words)
- Operand pointer (two words)

Environment data occupies 14 bytes when stored in memory. The format of environment data, when stored in memory with the FSTENV, or FNSTENV instructions, is shown on the following page.

Environment Data Format

← Increasing Addresses ↓

15 0	
CONTROL WORD	+ 0
STATUS WORD	+ 2
TAG WORD	+ 4
INSTRUCTION	+ 6
POINTER	+ 8
OPERAND	+10
POINTER	+12

Notes:

1. The instruction and operand pointers together form the "Exception Pointers" part of environment data. See Exception Pointers in a following section of this chapter.

2. +0,+2,+4...+12 are the address offsets of each word.

Save State

FSAVE/FNSAVE instructions writes the full 8087 state, (environment data plus the register stack) to the memory location defined by the destination operand. The format of the 94-byte save area is shown on the following page.

Save State Format

← Increasing Addresses ↓

15 0

	CONTROL WORD	+ 0
	STATUS WORD	+ 2
	TAG WORD	+ 4
	INSTRUCTION	+ 6
	POINTER	+ 8
	OPERAND	+10
	POINTER	+12
	SIGNIFICAND 15-0 ST(0)	+14
	SIGNIFICAND 31-16 ST(0)	+16
	SIGNIFICAND 47-32 ST(0)	+18
	SIGNIFICAND 63-48 ST(0)	+20
S	EXPONENT 14-0 ST(0)	+22
	SIGNIFICAND 15-0 ST(1)	+24
	SIGNIFICAND 31-16 ST(1)	+26
	SIGNIFICAND 47-32 ST(1)	+28
	SIGNIFICAND 63-48 ST(1)	+30
S	EXPONENT 14-0 ST(1)	+32
	Offsets +34 thru +82 would hold the contents of ST(2) thru ST(6) continuing the same format.	
	SIGNIFICAND 15-0 ST(7)	+84
	SIGNIFICAND 31-16 ST(7)	+86
	SIGNIFICAND 47-32 ST(7)	+88
	SIGNIFICAND 63-48 ST(7)	+90
S	EXPONENT 14-0 ST(7)	+92

15 14 0

Exception Pointers

The exception pointers consist of the instruction pointers and the operand pointers. Whenever the 8087 executes an instruction, it saves the instruction address and the instruction opcode in the exception pointers. If the instruction references a memory operand, the 8087 keeps the address of the operand.

An exception handler can be written to store these pointers in memory and get information about the instruction that caused the error. The format of the exception pointers (instruction and operand pointers) in memory, when stored with the FSTENV, FNSTENV, FSAVE, or FNSAVE instructions, is shown below.

Exception Pointer Format

Instruction Pointer (IP)

<table>
<tr><td colspan="2">15</td><td colspan="2">0</td></tr>
<tr><td colspan="4">IP (bits 15-0)</td></tr>
<tr><td colspan="2">IP-hi</td><td>0</td><td>OPCODE</td></tr>
<tr><td>15</td><td>12</td><td>11</td><td>10 0</td></tr>
</table>

Operand Pointer (OP)

<table>
<tr><td colspan="2">15</td><td>0</td></tr>
<tr><td colspan="3">OP (bits 15-0)</td></tr>
<tr><td colspan="2">OP-hi</td><td>0000 0000 0000</td></tr>
<tr><td>15</td><td>12</td><td>11 0</td></tr>
</table>

Notes:

1. IP-hi field is Instruction Pointer (bits 19-16).
2. OPCODE field is instruction opcode (bits 10-0), (bits 15-11) of opcode are always 11011 (binary).

3. The OP-hi field is the Operand Pointer (bits 19-16)

Register Stack

Each of the eight registers in the 8087's register stack is 80 bits wide, and each is divided into three fields: the sign, the exponent, and the significand fields. This format corresponds to the 8087's temporary real data type that is used for all calculations.

Register Stack

Reg	S	Exponent	Significand
R1			
R2			
R3			
R4			
R5			
R6			
R7			
R8			
	79	78 64	63 0

All numeric values kept in the 8087 registers are held in the 80-bit temporary real floating-point format, which is the same format as the 80-bit temporary real data type.

> **Note:** All data types are converted to the 80-bit temporary real data type when used by the 8087.

Two modes are available for addressing the stack registers. The first mode implicitly uses the top one or two registers. This mode does not require any addressing bits in the 8087 instruction. The other mode allows any other stack register to be used together with the top of stack register. The top of stack, or the other register, can be specified as the *destination*.

The TOP field in the status word identifies the current top-of-stack register. A load operation decrements TOP by 1 and loads a value into the new top-of-stack register. A store-and-pop operation stores the value from the current top of stack register and then increments TOP by 1.

> **Note:** If the TOP field in the status word = 000, a push operation, which decrements TOP, produces TOP = 111; similarly, popping the stack with TOP = 111 yields TOP = 000.

Like 8088 stacks in memory, the 8087 register stack grows down toward lower-addressed registers.

Explicit register addressing is top-relative where ST denotes the current stack top register and ST(*i*) refers to the *i*th register from ST in the stack. Where $0 \leq i \leq 7$, or ST(0) is the stack top and ST(7) is the stack bottom register. For example, if the TOP field in the status word contains 011, then register 3 is the top of stack. The following instruction would then add registers 3 and 5.

Example:

```
FADD ST,ST(2)
```

> **Note:** The register set in the 8087 and in the 8088 are in separate components. Direct transfer of values between the two register sets in one instruction is not possible. To transfer values between the 8087 and the 8088 register sets, the value must first pass through memory.

Status Word

The status word is a 16-bit register indicating the general condition of the 8087. The contents of the status word can be manually set, but are usually set by the 8087 instructions depending on the results. The status word can be examined by storing into memory with the 8087 FSTSW or FNSTSW instructions, and then inspecting memory with 8088 instructions.

The status word is divided into status bit and exception flag fields. The busy field (bit 15) shows whether the 8087 is executing an instruction (B=1) or is idle (B=0). The condition code is contained in C3 (bit 14) and C2-C0 (bits 10-8) of the status word. The TOP field (bits 13-11) of the status word point the the current stack top register. The IR field (bit 7) is the interrupt request bit indicating a pending interrupt to the 8088. Bits (5-0) are set to show that the 8087 has detected an exception while executing an instruction. The exception flags are: P,U,O,Z,D,I.

Status Word (high-byte)

15	14	13	12	11	10	9	8
B	C3	TOP			C2	C1	C0

For B values:

0 = 8087 idle (initialized state)
1 = 8087 busy

For TOP values:

000 = Register 0 is stack top (initialized state)
001 = Register 1 is stack top
010 = Register 2 is stack top
011 = Register 3 is stack top
100 = Register 4 is stack top
101 = Register 5 is stack top
110 = Register 6 is stack top
111 = Register 7 is stack top

For C3-C0 values after compare, or test instructions:

00x0 = ST > Source or 0
00x1 = ST < Source or 0
10x0 = ST = Source or 0
11x1 = ST is not comparable

Note: Position "x" (C1) not affected by these instructions. ST is the 8087 top of stack register.

For C3-C0 values after examine instructions:

0000 = Valid, positive, unnormalized
0001 = Invalid, positive, exponent = 0
0010 = Valid, negative, unnormalized
0011 = Invalid, negative, exponent = 0
0100 = Valid, positive, normalized
0101 = Infinity, positive
0110 = Valid, negative, normalized
0111 = Infinity, negative
1000 = Zero, positive
1001 = Empty
1010 = Zero, negative
1011 = Empty
1100 = Invalid, positive, exponent = 0
1101 = Empty
1110 = Invalid, negative, exponent = 0
1111 = Empty

Status Word (low-byte)

7	6	5	4	3	2	1	0
IR		P	U	O	Z	D	I

For IR values:

0 = No interrupt request (initialized state)
1 = Pending interrupt to the 8088

P,U,O,Z,D,I = Exception Flags

0 = No exception (initialized state)
1 = This exception condition has occurred

The 8087 checks for six classes of exception conditions during the execution of most instructions. The exceptions are:

1. P or Precision, for instructions that do exact arithmetic, this exception means that some precision has been lost in reporting the results of an operation. If the true result is not exactly representable in the *destination* format, the result is rounded according to the rounding mode defined by the RC field in the Control word, and the P flag is set.

2. U or Underflow, the true exponent is too small to be represented in the *destination* format. The result is non-zero but too small in magnitude to fit in the *destination* format.

3. O or Overflow, usually the exponent of the true result is too large for the *destination* real format. The result is too large in magnitude to fit the *destination* format.

4. Z or Zerodivide, division of a finite non-zero operand by zero.

5. D or Denormalized, an instruction attempts to operate on a denormalized number. At least one of the operands or the result is denormalized; it has the smallest exponent but a non-zero significand.

6. I or Invalid operations: Stack overflow, stack underflow, indeterminate form (0/0, ∞ - ∞, etc.) or the use of a Non-Number (NAN) as an operand. An exponent value is reserved and any bit pattern with this value in the exponent field is termed a Non-Number and causes this exception. Most invalid operations are the result of:

   - Programming errors such as trying to load a register that is not empty.

   - Popping an operand from a register that is empty.

   - Using operands that cause indeterminate results (for example, taking the square root of a negative number).

     Note: Refer to Special Values later in this chapter for a description of NAN, and other values.

When one of these six conditions occurs, the corresponding flag in the status word is set to 1. The flag remains "set" until reset, or cleared, by the FCLEX instruction, by initializing the 8087, or by overwriting the flags with an FRSTOR or FLDENV instruction. This means that the flags can provide a cumulative record of the exceptions encountered in a long calculation. A program can therefore mask all exceptions (except an invalid operation), run the calculation and then inspect the status word to see if any exceptions were detected at any point in the calculation.

Note: The 8087 has another set of "internal to the chip" exception flags that it clears before each instruction. These flags, and not those in the status word, trigger the 8087's exception response. The flags in the status word provide a cumulative record of exceptions for the programmer only.

Control Word

The control word is a 16-bit register that the user sets to determine the modes of computation the 8087 uses, and the error interrupts enabled. The control word consists of control fields, the interrupt enable mask, and the exception masks.

Control Word (high-byte)

15 13	12	11 10	9 8
reserved	IC	RC	PC

IC or Infinity Control (bit 12), two types of infinity arithmetic are provided:

0 = Projective	Is the initialized state, which treats both +infinity and -infinity as a single unsigned infinity.
1 = Affine	Is the state that respects both +infinity and -infinity.

Note: Projective mode is the recommended setting for most applications.

RC or Rounding control (bits 11 and 10), results are rounded in one of four directions:

00 = Round to nearest or even (initialized state)
01 = Round down (toward -∞)
10 = Round up (toward +∞)
11 = Chop (truncate toward zero)

PC or Precision control (bits 9 and 8), results are rounded to one of three precisions:

00 = 24 bits (Short Real)
01 = (reserved)
10 = 53 bits (Long Real)
11 = 64 bits (Temporary Real) (initialized state)

IEM or Interrupt-Enable Mask field (bit 7), when set to 1, all 8087 interrupts are masked or disabled.

The 8087 checks the appropriate exception mask field in the Control Word to determine if it should process the exception with a default handling procedure on the 8087 chip (mask=1) or invoke a user-written exception handler (mask=0). In the first case, the exception is said to be MASKED (from user software).

Control Word (low-byte)

7	6	5	4	3	2	1	0
IEM	x	PM	UM	OM	ZM	DM	IM

For IEM values:

0 = Interrupts enabled
1 = Interrupts disabled, masked (initialized state)

The exception mask fields are:

x = reserved
PM or Precision mask bit 5
UM or Underflow mask bit 4
OM or Overflow mask bit 3
ZM or Zerodivide mask bit 2
DM or Denormalized operand mask bit 1
IM or Invalid operation mask bit 0

For exception mask field values:

0 = Exception causes an interrupt that must be handled by a user-written interrupt handler.
1 = Exception is masked (initialized state)

The masked response to each one of the exceptions are:

P or Precision
Return rounded result and continue processing.

U or Underflow
The 8087 denormalizes (shift right) the fraction until the exponent is in range. This process is called gradual underflow.

O or Overflow
The 8087 generates coding for a properly signed infinity.

Z or Zerodivide
The 8087 generates coding for infinity signed with "exclusive or" of operand signs.

D or Denormalized
For a memory operand normal processing continues. For a register operand, convert to valid unnormal, then re-evaluate for exceptions.

I or Invalid operations
If one operand is NAN, return it; if both are NANs, return NAN with larger absolute value; if neither is NAN, return "indefinite".

Note: All exceptions are masked when the 8087 is in the initialized state.

When exceptions are masked, the 8087 can detect multiple exceptions in a single instruction, since it continues executing the instruction after doing its masked response. For example, the 8087 could detect a denormalized operand, do its masked response to this exception, and then detect an underflow.

By writing different values into the exception masks of the control word, the user can accept responsibility for handling exceptions, or delegate this to the 8087. Exception handling software is often difficult to write, and the 8087's masked responses have been tailored to deliver the most "reasonable" result for each condition. Most applications find that masking all exceptions other than "invalid operation" yield satisfactory results with the least programming effort. An invalid operation exception normally shows an unrecoverable error in a program that must be corrected; this exception is not normally masked.

If the 8087 executes an unmasked response to an exception, assume that a user-written exception handler is invoked via an interrupt from the 8087. The 8087 sets the IR (interrupt request) bit in the status word, but this, in itself, does not guarantee an immediate 8088 interrupt. The interrupt request can be blocked by the IEM (interrupt-enable mask) in the 8087 control word, or by the NMI (Non Mask Interrupt) Mask REG to the 8088.

The two methods of detecting exceptions are the NMI interrupt and polling the status word.

At power-on time, the NMI Mask REG is cleared to disable the NMI interrupt. This mask bit can be set and reset through system software as follows:

Set mask	Write hex 80 to I/O Address hex A0 (enable NMI).
Clear mask	Write hex 00 to I/O Address hex A0 (disable NMI).

Because a memory parity error can also cause an interrupt to the 8088 NMI line, the program should first check that a parity error did not occur (by reading the 8255 port).

> **Note:** If any exception flag is unmasked, it is imperative that the interrupt path to the 8088 is eventually cleared so the user's software can handle the exception and the offending task can resume execution.

User-written Exception Handler

A user-written exception handler takes the form of an 8088 interrupt procedure. Although exception handlers vary widely from one application to the next, most include these steps:

- Check the parity bits of the 8255 port to make sure that a memory parity error is not the cause of the interrupt.
- Store the 8087 environment as it existed at the time of the exception.
- Clear the exception bits in the status word.
- Enable interrupts on the 8088 (STI instruction).
- Identify the exception by examining the status and control words in the saved environment.

- Take application-dependent action.

- Return to the point of interruption, and resume normal execution.

Possible application-dependent actions include:

- Incrementing an exception counter for later display or printing.

- Printing or displaying diagnostic information, such as the 8087 environment and registers.

- Aborting further execution of the calculation causing the exception.

- Using the exception pointers to build an instruction that runs without exception and executing it.

- Storing a diagnostic value (NAN) in the result and continuing with the computation.

Notice that an exception may or may not constitute an error depending on the application. For example, an invalid operation caused by a stack overflow could signal an exception handler to extend the register stack into memory and continue running.

Tag Word

The Tag Word is a 16-bit register that is maintained by the 8087 showing the register stack contents. The Tag Word contains tags (2-bit position fields) indicating the contents of the corresponding stack registers.

For example: TAG (2) shows the contents of ST(2) stack register. A value of 0000000000110000B (0030H) in the Tag Word indicates that ST(2) is empty. A value of 0000000000010000B (0010H) in the Tag Word indicates the value in ST(2) is zero.

Tag Word (high-byte)

15 14	13 12	11 10	9 8
TAG(7)	TAG(6)	TAG(5)	TAG(4)

Tag Word (low-byte)

7 6	5 4	3 2	1 0
TAG(3)	TAG(2)	TAG(1)	TAG(0)

For tag values:

00 = Valid (Normal or Unnormal)

01 = Zero (True)

10 = Special (NAN, ∞, or Denormal)

11 = Empty (initialized state)

Data Types

The 8087 addresses seven different data types using all the 8088 addressing modes. All seven types are stored in memory with the sign bit located in the highest-addressed byte, and most significant digits of all numbers are the highest, or left-most digits.

Data Type	Bits	Significant Decimal Digits	Precision
Word integer	16	4-5	16 Bits
Short integer	32	9	32 Bits
Long integer	64	18	64 Bits
Packed decimal	80	18	18 Digits
Short real	32	6-7	24 Bits
Long real	64	15-16	53 Bits
Temporary real	80	19-20	64 Bits

Selecting Data Types

Each numeric data type is used for a specific function, they are:

Word integer	Index values, loop counts, and small program control values
Short integer	Large integer general computation
Long integer	Extended range integer computation
Packed decimal	Commercial and decimal conversion arithmetic
Short real	Reduced range and accuracy is traded for reduced memory requirements
Long real	Recommended floating-point variable type
Temporary real	Format for intermediate or high precision calculations

Data Formats

Binary Integers

The three binary integer formats are the same except for length, which governs the range that can be accommodated in each format. The leftmost bit in the following diagrams is interpreted as the number's sign: 0 = positive and 1 = negative. Negative numbers are represented in standard twos complement notation (the binary integers are the only 8087 formats to use twos complement). The quantity zero is represented with a positive sign (all bits are 0).

The word integer format is stored exactly like the 8088 16-bit signed integer, and is directly usable by instructions executed on either the 8087 or the 8088.

> **Note:** The 8088 does not provide single instruction support for the remaining numeric data types. Program instructions of the 8088 that read or write these data types must conform to the bit and byte order established by the 8087.

Binary Integer Formats

Word Integer

15	14	0
S	I14	I0

15	14	0
S	I14	I0

Short Integer

31	30	0
S	I30	I0

Long Integer

63	62	0
S	I62	I0

S = Sign bit (0=positive, 1=negative)
I = Integer bit

Decimal Integers

Decimal integers are stored into a 10-byte field with packed decimal notation. Decimal integers are stored with 2 decimal digits "packed" into each byte, except the leftmost byte, which only carries the sign bit (0 = positive, 1 = negative) in the high order position. Negative numbers are not stored in twos complement form and are distinguished from positive numbers only by the sign bit. The most significant digit of the number is the leftmost digit. All digits must be in the range 0H-9H.

Packed Decimal Format

Packed Decimal

79	78 72	71 64	63 56........15 8	7 0
S	x	D17 D16	D15 D14.....D3 D2	D1 D0

S = Sign bit (0=positive, 1=negative)
D = Decimal digit (two per byte)
x = These seven bit positions have no significance; the 8087 ignores them when loading, and zeros when storing.

Real Numbers

The 8087 stores real numbers in a three-field binary format that resembles scientific, or exponential, notation. The number's significant digits are held in the *significand* field, the *exponent* field locates the binary point within the significant digits (and therefore determines the number's magnitude), and the *sign* field shows whether the number is positive or negative. (The exponent and significand are analogous to the terms "characteristic" and "mantissa" sometimes used to describe floating-point numbers). Negative numbers differ from positive numbers only in their sign bits.

The 8087 carries the digits of the significand in normalized form. This means that, except for the value zero, the significand is an integer and a fraction as follows:

```
1^fff...ff
```

where ^ shows an assumed binary point. The number of fraction bits varies according to the real format: 23 for short, 52 for long and 63 for temporary real. By normalizing real numbers so that their integer bit is always a 1, the 8087 eliminates leading zeros in small values. This technique maximizes the number of significant digits that can be accommodated in a significand of a given width.

> **Note:** In the short and long real formats the integer bit is implicit and is not really stored; the integer bit is physically present in the temporary real format only.

If one were to examine only the significand with its assumed binary point, all normalized real numbers would have values between 1 and 2. The exponent field locates the actual binary point in the significant digits. Just as in decimal scientific notation, a positive exponent has the effect of moving the binary point to the right and a negative exponent effectively moves the binary point to the left, inserting leading zeros as necessary. An unbiased exponent of zero shows that the position of the assumed binary point is also the position of the actual binary point. The exponent field, then, determines a real number's magnitude.

To simplify comparing real numbers, the 8087 stores exponents in a biased form. This means a constant is added to the true exponent. The value of this bias is different for each real format. Refer to the following figures for the bias value for each format. It has been chosen to force the biased exponent to be a positive value. This allows two real numbers, of the same format and sign, to be compared as if they are unsigned binary integers. That is, when comparing them bit-wise from left to right beginning with the leftmost exponent bit, the first bit position that differs orders the numbers. No further comparison is needed. A number's true exponent can be determined simply by subtracting the bias value of its format.

The short and long real formats exist only in memory. If a number in one of these formats is loaded into a 8087 register, it is converted to temporary real, the format used for all 8087 internal operations. Likewise, data in 8087 registers can be converted to short or long real for storage in memory. The temporary real format can be used in memory also, typically to store intermediate results that cannot be held in 8087 registers.

Real Number Formats

Short Real

31	30 23	22 0
S	E7..E0	f1 f2......f22 f23

Long Real

63	62 52	51 0
S	E10.....E0	f1 f2 . . . . f51 f52

S = Sign bit (0=positive, 1=negative)
f*n* = Significand fraction bits
E = Biased Exponent bit
Exponent Bias (normalized values):
Short Real: 127 (7FH)
Long Real: 1023 (3FFH)

Notes:

1. The integer bit of significand is implicit in short and long real formats.
2. The implicit, (assumed) binary point is just to the left of the f1 bit-position.

Temporary Real Format

Temporary Real

79	78 64	63 62 0
S	E14........E0	I0 f1 f2 . . . . f62 f63

S = Sign bit (0=positive, 1=negative)
I0 = Integer bit of significand; stored in temporary real, implicit in short and long real formats.
f*n* = Significand fraction bits (1-63)
E = Biased Exponent bit
Exponent Bias (normalized values):
Temporary Real: 16383 (3FFFH)

Note: The implicit, (assumed) binary point is just to the left of the f1 bit-position.

Special Values

Besides being able to represent positive and negative numbers, the 8087 data formats can be used to describe other entities. These special values provide extra flexibility but most users do not need to understand them in detail to use the 8087 successfully.

The value zero can be signed positive or negative in the real and decimal integer formats; the sign of a binary integer zero is always positive. The fact that zero can be signed, however, is not apparent to the programmer.

The real number formats allow for the representation of the special values $+\infty$ and $-\infty$. The 8087 can generate these values as its built-in response to exceptions such as division by zero, or try to store a result that exceeds the upper range limit of the destination format. Infinities can participate in arithmetic and comparison operations, and in fact the processor provides two different conceptual models for handling these special values.

If a programmer attempts an operation for which the 8087 cannot deliver a reasonable result, it will at the programmer's discretion, either request an interrupt, or return the special value *indefinite*. Taking the square root of a negative number is an example of this type of invalid operation. The recommended action in this situation is to stop the computation by trapping to a user-written exception handler. If, however, the programmer elects to continue the computation, the specially coded *indefinite* value propagates through the calculation and thus flags the erroneous computation when it is eventually delivered as the result. Each format has coding that represents the special value *indefinite*.

In the real formats, a complete range of special values, both positive and negative, is designated to represent a class of values called NAN (Not-A-Number). The special value *indefinite* is a reserved NAN coding, but all other codings are made available to be defined in any way by application software. Using a NAN as an operand raises the invalid operation exception, and can trap to a user-written routine to process the NAN. Alternatively, the 8087's built-in exception handler simply returns the NAN itself as the result of the operation; in this way NAN's, including *indefinite*, can be propagated through a calculation and delivered as a final, special valued, result. One use for NANs is to detect uninitialized variables.

As mentioned earlier, the 8087 stores non-zero real numbers in normalized floating-point form. It also provides for storing and operating on real numbers that are not normalized, those significands contain one or more leading zeros. Non-normals arise when the result of a calculation yields a value too small to be represented in normal form. The leading zeros of non-normals allow smaller numbers to be represented, at the cost of some lost precision (the number of significant digits is reduced by the leading zeros). In typical algorithms, extremely small values are most likely generated as intermediate, rather than final results. By using the 8087's temporary real format for holding intermediates, values as small as $\pm 3.4 \times 10^{-4932}$ can be represented; this makes the occurrence of non-normal numbers a rare phenomenon in 8087 applications. Nevertheless, the 8087 can load, store and operate on non-normalized real numbers.

Data Format Requirements

The real number formats used by the 8087 are optimized for the processing of those numbers in the 8087 hardware. This format is sometimes called the IEEE format. The 8087 can only use real numbers that are in the 8087 real number format, therefore all real number data to be used by the 8087 must reside in memory in the proper 8087 format prior to use for input into the 8087. The real numbers output from the 8087 into memory are in the 8087 real number format.

The real number formats used by the 8088 are optimized for the processing of those numbers with software. This format is the Microsoft Binary real format.

8087 Real Number Formats

The data formats are described reading from the byte with the highest address or most significant byte to the the byte with the lowest address or least significant byte.

The short real number format is:

- Sign bit in the upper bit of the first byte.
- Exponent (8 bits) in the remaining 7 bits of the first byte and the upper bit of the second byte.
- Significand (23 bits) in the remaining 7 bits of the second byte and all bytes 3 and 4.
- Exponent bias: 127 (7FH).

The long real number format is:

- Sign bit in the upper bit of the first byte.
- Exponent (11 bits) in the remaining 7 bits of the first byte, and the upper 4 bits of the second byte.
- Significand (52 bits) in the remaining 4 bits of the second byte and all bytes 3 through 8.
- Exponent bias: 1023 (3FFH).

In short real (hex notation, bytes high to low):

```
3F800000 =  1
BF800000 = -1
```

In short real (hex notation, bytes low to high similar to reading DEBUG listing):

```
0000803F =  1
000080BF = -1
```

In long real (hex notation, bytes high to low):

```
3FF0000000000000 =  1
BFF0000000000000 = -1
```

In long real (hex notation, bytes low to high similar to reading assembler or DEBUG listings):

```
000000000000F03F =  1
000000000000F0BF = -1
```

If the exponent is equal to zero then the number is equal to zero. If the exponent is not equal to zero then the number is:

```
<NUMBER>=<SIGN>*1.<SIGNIFICAND>B*2**(<EXPONENT>-BIAS)
```

If the exponent is equal to FFH for short real or 7FFH long real, then an error occurs.

Note: Refer to the previous section on real numbers for more detail.

Microsoft Binary Real Format

The data formats are described reading from the byte with the highest address or most significant byte to the the byte with the lowest address or least significant byte.

The single precision real number format is:

- Exponent (8 bits) in the first byte.
- Sign bit in the upper bit of the second byte.
- Significand (23 bits) in the remaining 7 bits of the second byte and all bytes 3 and 4.
- Exponent bias: 129 (81H).

The double precision real number format is:

- Exponent (8 bits) in the first byte.
- Sign bit in the upper bit of the second byte.
- Significand (55 bits) in the remaining 7 bits of the second byte and all bytes 3 through 8.
- Exponent bias: 1025 (401H).

Single Precision

31 24	23	22 0
E7....E0	S	f1 f2......f22 f23

Double Precision

63 56	55	54 0
E7....E0	S	f1 f2 . . . . f54 f55

S = Sign bit (0=positive, 1=negative)
f*n* = Significand fraction bits
E = Biased Exponent bit
Exponent Bias (normalized values):
Short Real: 129 (81H)
Long Real: 1025 (401H)

Notes:

1. The integer bit of significand is implicit.
2. The implicit, (assumed) binary point is just to the left of the f1 bit-position.

In single precision (hex notation, bytes high to low):

```
81000000 =  1
81800000 = -1
```

In single precision (hex notation, bytes low to high similar to reading DEBUG listing):

```
00000081 =  1
00008081 = -1
```

If the exponent is equal to zero then the number is equal to zero. If the exponent is not equal to zero then the number is:

```
<NUMBER>=<SIGN>*.1<SIGNIFICAND>B*2**(<EXPONENT>-80H)
```

> **Note:** An excellent method of looking at the different formats is to write an assembler source test file with define constants for each of the different formats with various data. Assemble the source file creating a listing; observe how each number was assembled in the hex portion of the listing. Use the /R parameter and assemble the same source test file. Notice the difference in the data formats and values. A normal assembly creates Microsoft Binary data formats and the /R parameter tells the assembler to use 8087 data formats and assemble the 8087 floating-point instructions.

Data Format Conversion

Data format conversion is necessary when the data destined for 8087 input is not in 8087 format, or 8087 data output is destined for 8088 use or output to any output device. This means that a real number entered from IBM Personal Computer Basic, Compiler Basic, version 1.0 Fortran or Pascal cannot be directly moved into the 8087 and used. Numbers must be converted from Binary real format to 8087 and then, after all computations are finished, converted back from 8087 to Binary real format.

Data format conversion is not necessary when using IBM Personal Computer FORTRAN or Pascal real number data formats version 2.0.

The conversion process from Binary real single precision to 8087 short real can be accomplished one of many ways. One way is to save the sign bit of the number, shift the first byte right one bit with the lowest bit being placed into the upper bit of the second byte, and then placing the sign bit into bit 7 of the first byte. Because the biasing scheme is different for these formats, subtract two from the exponent. Reverse the operation when going from 8087 short real back to Binary real single precision.

A similar process would be required for the conversion of Binary real double precision to 8087 long real and back again.

Data Conversion routines are included for the conversion process or your own routines can be used.

Data Conversion Routines

Enclosed on the Macro Assembler diskette are the following files:

IBMUTIL.LIB
IBMASM.ASM
UTEST.PAS

If the PASCAL program is linked with IBMUTIL and the REGMATH library, then the input/output IBMUTIL conversion routines are used instead of the ones out of the REGMATH library. (They are interchangeable with the REGMATH versions included in the IBM Personal Computer Pascal 2.0, and FORTRAN 2.0.)

Since no segment references are in the routines, they are relocatable. They also use an 8-byte stack. The six conversion routines work within the registers and the passed arguments. Only the input/output conversions use data areas.

The following assumptions are made by the input/output routines. (These routines are the same as those used in the FORTRAN 2.0 and Pascal 2.0 REGMATH libraries.)

Using The ASCII - 8087 Conversion Routines

Assumptions:

- All code is in a segment with the following segment declaration:

  ```
  MATHCODE SEGMENT BYTE PUBLIC 'CODE'
  MATHCODE ENDS
  ```

- All static data used by this routine is in a segment with the following segment declaration:

  ```
  DATA    SEGMENT WORD PUBLIC 'DATA'
  DATA    ENDS
  DGROUP  GROUP   DATA
  ```

- All the routines are near procedures and must be called with intra-segment CALLs or JMPs:

- The DS and ES registers are equal.

- All non-segment registers except SP are destroyed unless used to return results.

$I8_INPUT - Floating-Point and Integer Numeric Input

This routine converts ASCII strings into 8087 format floating-point numbers. This routine does not handle denormal, infinity, or NAN inputs. This routine accepts BASIC, FORTRAN, and PASCAL type number strings.

Calling Sequence:

```
MATHCODE SEGMENT BYTE PUBLIC 'CODE'
         EXTRN   $I8_INPUT:near
         (set up registers)
         CALL    $I8_INPUT
         (results returned)
MATHCODE ENDS
```

Inputs:

SI:DS	Offset of character stream to analyze for the value.
DI:DS	Offset of space for 8 byte double precision result.
AX	FORTRAN scale factor (AX = 0 for normal use).
BX	Radix desired (1..36), integer only to be returned (BX = 0 for floating-point input).
CX	Maximum character count for scanning input stream.

Outputs:

SI:DS	Offset of first byte after decoded string value.
DI:DS	Offset of 8 byte result
AX:	Low order part of integer value (32 bit integer).
BX:	High order part of integer value (32 bit integer).
CL:	Return Flags (bit set to 1 if condition is true). (See below for bit definitions.)

128	String of decimal points decoded to indefinite value.
64	No digits found or syntax error, number is invalid.
32	32-bit signed value is invalid.
16	32-bit unsigned value is invalid.
8	CL(2) OR there were more than 7 digits in the number (Basic DP).
4	An explicit ’D’, or ’d’ was found for exponent (FORTRAN DP).
2	An explicit ’.’, ’E’, ’e’, ’D’, or ’d’ was found (FORTRAN Real).
1	DP overflow (too big for DP Format).

$I8__OUTPUT - Floating-Point Numeric Output

This routine converts double precision 8087 format floating-point numbers into ASCII strings. This routine does not handle denormal, infinity, or NAN inputs.

Calling Sequence:

```
MATHCODE SEGMENT BYTE PUBLIC 'CODE'
         EXTRN   $I8_OUTPUT:near
         (set up registers)
         CALL    $I8_OUTPUT
         (results returned)
MATHCODE ENDS
```

Inputs:

SI:DS	Offset address of 8 byte double precision number.

Outputs:

SI:DS	Offset address of LSTRING(16),e.g. 17 byte area with length in 1st byte followed by a string of digits (decimal point on left); a non-zero number has no leading or trailing zeros. A zero number has a single zero with an exponent of zero.
AX	1 if original number was OK, 0 if it was indefinite.
BL	Sign character; blank for zero or positive, "-" for negative.
DX	Base ten exponent.

Using The Microsoft - 8087 Conversion Routines

Assumptions:

- All code is in a segment with the following segment declaration:

```
MATHCODE SEGMENT BYTE PUBLIC 'CODE'
MATHCODE ENDS
```

- All the routines are near procedures and must be called with intra-segment CALLs or JMPs.

- All non-segment registers except SP are destroyed unless used to return results.

The conversion routines are:

```
$I4_M4 - Convert Microsoft single to 8087 single
$I8_M8 - Convert Microsoft double to 8087 double

$M4_I4 - Convert 8087 single to Microsoft single
$M8_I8 - Convert 8087 double to Microsoft double

$I4_I8 - Convert 8087 double to 8087 single
$I8_I4 - Convert 8087 single to 8087 double
```

Purpose:

These routines convert between Microsoft and 8087 formats for both single and double precision. See the above headings for format and types of the inputs and outputs.

If no errors occur, then AX=0 and the carry flag is clear. If any errors occur, then AX=1 and the carry flag is set.

These routines do not handle 8087 denormal, infinity, or NAN inputs.

Calling Sequence:

```
MATHCODE SEGMENT BYTE PUBLIC 'CODE'
         EXTRN   $I4_M4:NEAR
         (set up registers)
         CALL    $I4_M4
         (results returned)
MATHCODE ENDS
```

Inputs:

DS:SI	Address of input number in 8087 or Microsoft format.
ES:DI	Address for result in the other format.

Outputs:

SI	Address of result (maybe invalid if error).
DI	Address of result (maybe invalid if error).
AX	0 if no error , 1 if error.
F	'C' clear if no error, 'C' set if error.

Processor Control and Monitoring

Co-processing

The 8087 and host 8088 act as co-processors. They share the same instruction stream and sometimes carry out parallel executions. The 8088 has a set of ESCAPE instructions that, in memory addressing mode, cause the 8088 to calculate the address and read the contents of that address. The 8088 ignores the word it reads and executes the next instructions. The 8087, however, monitors the same instruction stream and when it detects an ESCAPE it begins processing. The 8087 decodes the instruction to determine how many more words it needs from memory. It increments the address and fetches data until all required data is read. The 8087 then releases the bus and begins calculating while the 8088 continues executing the instruction stream.

The 8088 WAIT instruction allows software to synchronize the 8088 to the 8087 so that the 8088 does not execute the next instruction until the 8087 is finished with its current (if any) instruction. To do this, the programmer codes the FWAIT instruction immediately before an 8088 instruction that accesses a memory operand read or written by a previous 8087 instruction.

Initialization

The 8087 can be initialized by hardware or software. Hardware initialization (reset) occurs when powering on or doing a reset by pressing Ctrl-Alt-Del keys. The FINIT (initialize) and FSAVE (save state) instructions do a software initialization of the 8087. The state of the 8087 following initialization is shown in the table below.

Field	Value	Interpretation
Control Word		
Infinity Control	0	Projective
Rounding Control	00	Round to nearest
Precision Control	11	64 bits
IEM	1	Interrupts disabled
Exception Masks	111111	All exceptions masked
Status Word		
Busy	0	Not busy
Condition Code	????	(Indeterminate)
TOP	000	Empty stack
Interrupt Request	0	No interrupt
Exception Flags	000000	No exceptions
Tag Word		
Tags	11	Empty
Exception Pointers		
Opcode	N.C.	Not changed
Instruction Add.	N.C.	Not changed
Operand Add.	N.C.	Not changed
Registers	N.C.	Not changed

Hardware initialization causes the 8087 to begin tracking the 8088 instruction fetches and execution. Initialization does not affect the content of the registers or of the exception pointers (these have indeterminate values immediately following power up). However, since the stack is effectively emptied by initialization (TOP = 0, all registers tagged empty), the contents of the registers should normally be considered destroyed by initialization.

Interrupts and Waits

Interrupt Requests

All interrupt requests originate with the detection of an exception. The interrupt request is made if the exception is unmasked and 8087 interrupts are enabled; that is both the relevant exception mask and the interrupt-enable mask are clear (0). If the exception is masked, the 8087 executes its masked response and does not set the interrupt request bit.

If the exception is unmasked but interrupts are disabled (IEM = 1), the 8087's action depends on whether the 8088 is waiting (the 8087 "knows" if the 8088 is waiting because it decodes the WAIT instruction in parallel with the 8088). If the 8088 is *not* waiting, the 8087 assumes that the 8088 does not want to be interrupted now and that it will enable interrupts on the 8087 when it does. The 8087 sets the interrupt request bit and holds its BUSY line active. The 8087 CU continues to track the 8088, and if an 8087 instruction (without a preceding WAIT) comes along, it is executed. Normally in this situation the instruction is FNENI (enable interrupts without waiting). This clears the interrupt-enable mask and the 8087 then activates INT, (the interrupt line to the 8088). However, any instruction will be executed, and it is therefore conceivably possible to abort the interrupt request before it is ever handled. Aborting an interrupt request in this manner, however, would normally be considered a program error.

The interrupt to the 8088 goes through the NMI (non-maskable interrupt) line of the 8088. At power-on time, the NMI Mask REG is cleared to disable the NMI. This mask bit can be set and reset through system software as follows:

Set mask	Write hex 80 to I/O Address hex A0 (enable NMI).
Clear mask	Write hex 00 to I/O Address hex A0 (disable NMI).

If the 8088 is waiting, then the processors are in danger of entering an endless wait condition. To prevent this condition, the 8087 *ignores* the fact that interrupts are disabled and activates INT even though the interrupt-enable mask is set.

The interrupt request bit remains set until it is explicitly cleared (if INT is not disabled by IEM, it remains active also). This can be done by the FNCLEX, FNSAVE, or FNINT instructions. The interrupt procedure that handles the 8087's interrupt request, the exception handler, must clear the interrupt request bit before returning to normal execution on the 8087. If it does not, the interrupt is immediately generated again and the program enters an endless loop.

Endless Wait

The 8087 and the 8088 can enter an endless wait condition when the 8088 is executing a WAIT instruction and a pending interrupt request from the 8087 is prevented from being recognized by the 8088. Thus, the 8088 waits for the 8087 to lower its BUSY line, while the 8087 waits for the 8088 to invoke the exception handler interrupt procedure, and the task that has generated the exception is blocked from further execution.

A user-written exception handler can itself cause an unending wait. When the exception handler starts to run, the 8087 is suspended with its BUSY line active, waiting for the exception to be cleared, and interrupts on the 8088 are disabled. If, in this condition, the exception handler issues any 8087 instruction, other than in a no-wait form, the result is an unending wait. To prevent this, the exception handler should clear the exception on the 8087 and enable interrupts on the 8088 before executing any instruction preceded by a WAIT.

An instruction preceded by a WAIT (or an FWAIT instruction) should never be executed when 8088 interrupts are disabled and any possibility exists that the 8087's BUSY line is active.

Appendixes

Contents

Appendix A. Messages **A-3**

Assembler Messages **A-3**
  Small Assembler (ASM) A-3
  Macro Assembler (MASM) A-3

I/O Handler Messages **A-12**

Runtime Messages **A-14**

Library Manager Error Messages **A-15**
  Unrecoverable Errors A-15
  Other Errors A-17

LINK Messages **A-18**

SALUT Error Messages **A-24**

Appendix B. Appendix B. Instruction Set Summary B-1

8088 Instruction Set Summary **B-1**

8087 Instruction Set Summary **B-16**

80286 Instruction Set Summary **B-27**

Appendix C. Pseudo Operations Summary **C-1**

Conditional Pseudo Operations **C-2**

Data Pseudo Operations **C-3**

Listing Pseudo Operations **C-5**

Macro Pseudo Operations **C-6**
  Special Macro operators C-6

Mode Pseudo Operations **C-6**

Appendix D. The Sample Sessions **D-1**

Introduction **D-1**
  Requirements D-3

TRYCOM Sample Sessions **D-4**
  Small Assembler Session D-5
    Small Assembler Samples Diskette D-5
    Small Assembler Sample Procedure D-6
    Small Assembler Session Summary D-9
  Macro Assembler Session D-10
    Macro Assembler Samples Diskette D-10
    Macro Assembler Sample Procedure ... D-10

SALUT Sample Session **D-12**
    SALUT Samples Diskette D-14
    SALUT Sample Procedure D-16
    SALUT Sample Session Summary D-22

Assembler Source Coding Aids **D-23**
    Skeletal Assembler Source D-23
    8087 Assembler Sample Source. D-24

Appendix A. Messages

Assembler Messages

Small Assembler (ASM)

Only the error code is displayed indicating that you have an error in your program. Refer to the codes listed in the following paragraphs for a description of the error messages.

Macro Assembler (MASM)

If you have any errors in your assembly program, the assembler tells you in one of two ways:

- The error message is inserted into the listing file.
- The screen displays the error message.

Carefully check and correct your *source* program of all types of errors.

After you have corrected your program, assemble your program again.

Block nesting error (Code 0)
Nested procedures, segments, structures, macros, IRC, IRP, or REPT are not properly terminated. An example of this error is closure of an outer level of nesting with inner levels still open.

Extra characters on line (Code 1)

This occurs when enough information to define the instruction directive is on a statement line, but superfluous characters beyond are also present.

Register already defined (Code 2)

This only occurs if the assembler has internal logic errors.

Unknown symbol type (Code 3)

Symbol statement has something unrecognizable in the type field.

Redefinition of symbol (Code 4)

This error occurs on pass 2 and succeeding definitions of a symbol.

Symbol is multi-defined (Code 5)

This error occurs on a symbol that is later redefined.

Phase error between passes (Code 6)

The program has ambiguous instruction directives so the location of a label in the program changed in value between pass 1 and pass 2 of the assembler. For example, a forward reference coded without a segment override where one is required. There would be an additional byte (the code segment override) generated in pass 2 causing the location of the next label to change. You can use the /D parameter to produce a listing to aid in resolving phase errors between passes (see "Parameters" in Chapter 5 "MACRO Assembler").

Already had ELSE clause (Code 7)

Attempt to define an ELSE clause within an existing ELSE clause.

Not in conditional block (Code 8)

An ENDIF or ELSE is specified without a previous conditional assembly directive active.

Symbol not defined (Code 9)
A symbol is used that has no definition.

Syntax error (Code 10)
The syntax of the statement does not match any recognizable syntax.

Type illegal in context (Code 11)
The type specified is of an unacceptable size.

Should have been group name (Code 12)
Expecting a group name but something other than this was entered.

Must be declared in pass 1 (Code 13)
Assembler expecting a constant value but got something else. An example of this might be a vector size being a forward reference.

Symbol type usage illegal (Code 14)
Illegal usage of a PUBLIC symbol.

Symbol already different kind (Code 15)
Attempt to define a symbol differently from a previous definition.

Symbol is reserved word (Code 16)
Attempt to use an assembler reserved word illegally (for example, to declare MOV as a variable).

Forward reference is illegal (Code 17)
Attempt to forward reference something that must be defined in pass 1.

Must be register (Code 18)
Register expected as operand but user furnished symbol was not a register.

Wrong type of register (Code 19)

Directive or instruction expected one type of register and another was specified. For example, ASSUME AX instead of ASSUME *seg-reg*

Must be segment or group (Code 20)

Expecting segment or group and something else was specified.

Symbol has no segment (Code 21)

Trying to use a variable with SEG and the variable has no known segment.

Must be symbol type (Code 22)

Must be WORD, DW, QW, BYTE, or TB but received something else.

Already defined locally (Code 23)

Tried to define a symbol as EXTERNAL that had already been defined locally.

Segment parameters are changed (Code 24)

List of arguments to SEGMENT were not identical to the first time this segment was used.

Not proper align/combine type (Code 25)

SEGMENT parameters were incorrect.

Reference to mult defined (Code 26)

The instruction references something that is multi-defined.

Operand was expected (Code 27)

Assembler is expecting an operand but something else was received.

Operator was expected (Code 28)

Assembler was expecting an operator but something else was received.

Division by 0 or overflow (Code 29)
An expression is given that results in a divide by 0.

Shift count is negative (Code 30)
A shift expression is generated that results in a negative shift count.

Operand types must match (Code 31)
Assembler gets different kinds or sizes of arguments in a case where they must match. For example, exchange.

Illegal use of external (Code 32)
Usage of an external in some illegal manner. For example, DB M DUP(?) where M is declared external.

Must be record field name (Code 33)
Expecting a record field name but got something else.

Must be record or fieldname (Code 34)
Expecting a record name or field name and received something else.

Operand must have size (Code 35)
Expecting the size of the operand but received something else.

Must be var, label or constant (Code 36)
Expecting a variable, label, or constant but received something else.

Must be structure field name (Code 37)
Expecting a structure field name but received something else.

Left operand must have segment (Code 38)
Used something in right operand that required a segment in the left operand. For example, ":".

One operand must be const (Code 39)
This is an illegal use of the addition operator.

Operands must be same or 1 abs (Code 40)
Illegal usage of the subtraction operator.

Normal type operand expected (Code 41)
Received STRUC, FIELDS, NAMES, BYTE, WORD, or DW when expecting a variable label.

Constant was expected (Code 42)
Expecting a constant an received something else.

Operand must have segment (Code 43)
Illegal use of SEG directive.

Must be associated with data (Code 44)
Use of code related item where data related item was expected. For example, DS override of a procedure.

Must be associated with code (Code 45)
Use of data related item where code related item was expected.

Already have base register (Code 46)
Trying to double base address.

Already have index register (Code 47)
Trying to double index address.

Must be index or base register (Code 48)
Instruction requires a base or index register and some other register was specified.

Illegal use of register (Code 49)
Usage of a register with an instruction where there is no 8088 instruction possible.

Value is out of range (Code 50)
Value is too large for expected use. For example, moving a DW to a register.

Operand not in IP segment (Code 51)

Access of operand is impossible due to not being in the current IP segment.

Improper operand type (Code 52)

Use of an operand such that the op code cannot be generated.

Relative jump out of range (Code 53)

Relative jumps must be within the range -128 +127 of the current instruction and the specified jump is beyond this range.

Index displ. must be constant (Code 54)

Trying to use a variable displacement off an index register.

Illegal register value (Code 55)

The register value specified does not fit into the "reg" field (the reg field is greater than 7).

No immediate mode (Code 56)

Immediate mode specified or an op code that cannot accept the immediate. For example, PUSH.

Illegal size for item (Code 57)

Size of referenced item is illegal. For example, shift of a double word.

Byte register is illegal (Code 58)

Use of one of the byte registers in context where it is illegal. For example, PUSH AL.

CS register illegal usage (Code 59)

Trying to use the CS register illegally. For example, XCHG CS,AX.

Must be AX or AL (Code 60)

Specification of some register other than AX or AL where only these are acceptable. For example, the IN instruction.

Improper use of segment reg. (Code 61)

Specification of a segment register where this is illegal. For example, an immediate move to a segment register.

No or unreachable CS (Code 62)

Trying to jump to an unreachable label.

Operand combination illegal (Code 63)

Specification of a two-operand instruction where the combination specified is illegal.

Near JMP/CALL to different CS (Code 64)

Attempt to do a NEAR jump or call to a location in a different code segment.

Label can't have seg. override (Code 65)

Illegal usage of segment override.

Must have opcode after prefix (Code 66)

Use of one of the prefix instructions without specification of the correct op code (or any op code) it is expecting.

Can't override ES segment (Code 67)

Trying to override the ES segment in an instruction where this override is not legal. For example, store string.

Can't reach with segment reg (Code 68)

There is no ASSUME that makes the variable reachable.

Must be in segment block (Code 69)

Attempt to generate code when not in a segment.

Can't use EVEN on BYTE segment (Code 70)

Segment was declared to be byte segment and attempt to use EVEN was made.

Forward needs override (Code 71)

Forward needs override or FAR.

Illegal value for DUP count (Code 72)

DUP counts must be a constant that is not 0 or negative.

Symbol already external (Code 73)

Attempt to define a symbol as local that is already external.

DUP is too large for linker (Code 74)

Nesting of DUPs was such that too large a record was expected from the linker.

Usage of ? (indeterminate) bad (Code 75)

Improper usage of the "?". For example, ?+5.

More values than defined with (Code 76)

Only initialize list legal (Code 77)

Directive illegal in STRUC (Code 78)

Override with DUP is illegal (Code 79)

Field cannot be overridden (Code 80)

Override is of+ wrong type (Code 81)

Register can't be forward ref (Code 82)

Circular chain of EQU aliases (Code 83)

(Code 84)

Feature not supported by the Small Assembler (ASM) Use MASM to assemble or remove the unsupported feature from your source.

8087 opcode can not be emulated, produced by MASM. The opcode cannot be used with emulation mode.

End of file, no END directive (Code 85)

Either the end of file was reached prematurely, or you did not use the END pseudo-op. Possibly, missing ENDM or conditional assembly statement, ENDIF. Also, may have something between a LOCAL and MACRO statement.

I/O Handler Messages

This set of messages is generated by the I/O handlers. You see the message in one of the following forms:

```
ASM Error -- ###
    in: filename

MASM Error --error message text
     in: filename
```

Note: *filename* is the name of the file associated with the message.

The error messages are completed by *error-message-text* or 3-digit code (in ###) from this list:

Hard data (Code 101)

Device name (Code 102)

Operation (Code 103)

File system (Code 104)

Device offline (Code 105)

Lost file (Code 106)

File name (Code 107)

Device full (Code 108)

Unknown device (Code 109)

File not found (Code 110)

Protected file (Code 111)

File in use (Code 112)

File not open (Code 113)

Data format (Code 114)

Line too long (Code 115)

Runtime Messages

In addition to the above Assembler messages, the following messages can be displayed as you run your program:

Out of Memory

This message has no corresponding number. Either the source is too big or too many labels are in the symbol table. If an "Out of Memory" condition occurs with either assembler, there is insufficient work space to continue. To recover from such a condition, try rerunning the assembler, but only specify one file. On one run, get the listing; on another, the OBJ file. Some items that use up work space are: open INCLUDE files, many symbol names, long symbol names, open OUTPUT files, defined STRUCs and MACROS (MACRO Assembler only). Avoiding these allow much longer assemblies to be done with the minimum of work space available. In the MACRO Assembler, space can be regained by a PURGE of a MACRO after invoking it for the last time before defining another MACRO. In MACRO definition, be sure to separate fields with TAB characters, not a series of blanks, to conserve space. Avoid single semicolon comments in macros; use double semicolon comments instead. Additional memory increases work space, but neither assembler uses the memory above that available in a 196K RAM machine.

Internal Error

Usually caused by an arithmetic check. If it occurs, notify your authorized IBM Personal Computer Dealer.

Library Manager Error Messages

Unrecoverable Errors

Aborted by user
User did not want to create a library.

Alignment factor too small
Library larger than (page size)*64K.

Cannot create new library
Directory full or name is bad.

Cannot open VM.TMP
Directory is full.

Cannot read from VM
An error in the Library Manager.

Cannot rename old library
Bad file name.

Cannot reopen library
An error in the Library Manager.

Cannot write to VM
Disk is full.

Command syntax error
Typographical error.

Error writing to cross reference file
Disk is full.

Error writing to new library
Disk is full.

Fetch: not allocated
An error in the Library Manager.

Free: not allocated
An error in the Library Manager.

Insufficient memory.
Not enough memory for Library Manager.

Internal failure
An error in the Library Manager.

Invalid file name extension
Extension is neither ".OBJ" nor ".LIB".

Invalid library
File is not a library file.

Invalid library name extension
Name does not end in ".LIB".

Invalid object module . . .
Object module contains damaged data.

Mark: not allocated
An error in the Library Manager.

No more virtual memory
Too many public symbols.

Syntax error
Typographical error.

Syntax error (bad file spec)
Typo specifying file name.

Syntax error (bad input)
Illegal character in input.

Syntax error (switch name expected)
Name not found after "/".

Syntax error (switch value expected)
Illegal character after ":".

Too many symbols
Too many public symbols.

Unexpected EOF on command input
Unexpected EOF on response file.

Unknown switch
Unknown name after "/".

Write to extract file failed
Disk is full.

Write to library file failed
Disk is full.

Other Errors

Cannot create extract file . . .
Directory is full.

. . . extension illegal--file ignored
Extension is neither ".LIB" nor ".OBJ".

. . . invalid format . . . ; file ignored
Input file is neither object or library.

. . . invalid library header
Input file is not really a library.

. . . module not in library; ignored
Module to extract or delete nonexistent.

. . . page size too small--ignored
Value of page size switch too small.

LINK Messages

The following messages are in alphabetic order.

About to generate .EXE file
Change disks <hit ENTER>

Informational message. This message is displayed when you specify the /PAUSE parameter. Insert your Runfile diskette into the appropriate drive and press Enter.

An internal failure has occurred

An error occurred in the linker program. Write down the conditions under which the message appeared. Report this to your Authorized IBM Personal Computer Dealer.

Attempt to access data outside of segment bounds

An object file is invalid. Review the .ASM file or assembled listing for segmentation violations. Look for a bad reference or invalid instruction.

Big bit not implemented

The Linker does not support segments exactly 64K bytes long. Reduce the size of the offending segment.

Cannot find file ***object file***
Change diskette <hit ENTER>

The linker could not locate the specified object module on the drive. Insert the correct diskette with the *object file* on it and press Enter.

> **Warning:** If it is necessary to change the diskette that contains the open VM.TMP file, you need to exit LINK by pressing Ctrl-Break instead of changing diskettes, and then restart LINK with a different drive specified to locate the object file. Otherwise,

there can be a loss of data on the diskette inserted. This usually occurs when using default drive values for the object file.

Cannot find library *library file*
Enter new drive letter:

The specified library could not be found on the drive. Enter the correct letter for the drive the library is on.

Warning: If it is necessary to change the diskette that contains the open VM.TMP file, you need to exit LINK by pressing Ctrl-Break instead of changing diskettes, and then restart LINK with a different drive specified to locate the object file. Otherwise, there can be a loss of data on the diskette inserted. This usually occurs when using default drive values for the object file.

Cannot nest response file

You used @*filespec* within an automatic response file. Automatic response files cannot be nested. Change the initial auto response file to eliminate nested auto response file. Or, fix the syntax error if you did not intend this to be an auto response file.

Cannot open list file

The directory or disk is full. Insert another disk, or delete some files from the disk that is full.

Cannot open overlay

The directory or disk is full. Insert another disk, or delete some files from the disk that is full.

Cannot open response file

The automatic response file could not be found. Include drive specifier and/or path for the response file. Place file on proper disk.

Cannot open temporary file

The directory or disk is full. Insert another disk, or delete some files from the disk that is full.

DUP record too complex

The problem is caused by too many structures or DUP statements in the object module created from an assembler source program. The most likely one is nested too deeply. Reduce the number of structures or DUP statements in the assembler source program, create a new object module, and retry LINK.

Fixup offset exceeds field width

An assembler instruction refers to an address with a NEAR attribute instead of a FAR attribute, or the size of a group exceeds 64K (See .LST file). An explicit override specifying a SEG REG may have contradicted the usage of that SEG REG as defined by the ASSUME pseudo-op. Edit assembler source program and process again.

Invalid format file

A library is in error. Restore library file from your backup disk and try again.

Bad numeric parameter

The numeric value is not in digits. Run LINK again and name the numeric parameter with values of 0 through 9 for each digit.

Invalid object module

Object module was incorrectly formed or unobserved errors occurred during compilation. The disk may be bad. Recompile the module to either the same or a different disk.

Invalid switch

The character indicated on the preceding line is not a valid linker parameter (switch). Review Chapter 9, "The Linker (Link) Program" to determine which characters are a valid link parameters (switches).

No object modules specified

You did not name any object modules in the command line or in response to the prompt. Name the object modules, since the linker needs some files to link.

Out of space on list file

Insufficient disk space for the List file. Use a disk with enough free space to hold the file.

Out of space on run file

This error usually occurs when there is not enough disk space for the Run file (.EXE). Use a disk with enough free space to hold the file.

Out of space on VM.TMP

No more disk space remained to expand the VM.TMP file. Use a disk with enough free space to hold this file.

Program size exceeds capacity of LINK

Load module is too large for processing. Reduce the size of your program.

Reloc table overflow

Program contains more than 12000 long calls. Reduce the number of long calls. If DEBUG was on in a Pascal or Fortran program, turn DEBUG off, recompile then re-link.

Segment size exceeds 64K

This message indicates that you attempted to combine identically named segments that resulted in a segment requirement of greater than 64K bytes. You cannot address more than 64K bytes. Change segment names in object modules and try link again.

Stack size exceeds 65535 bytes

Informational message. The size specified for the stack must be less than or equal to 65535. No action required.

Symbol defined more than once

The Linker found two or more modules that define a single symbol name. Check for the following:

- Are you sure you're not linking the same files twice?
- Does one of the linked modules have a symbol incorrectly identified as *public* instead of *external*?

Symbol table overflow

Many, very long names were entered. The names exceeded approximately 50K bytes. Use shorter and/or fewer names.

There was/were *number* errors detected

This message is displayed for your information at the end of the link session. No action required.

Too many external symbols in one module

The limit is 256 external symbols per module. This message indicates that the limit was exceeded. Break up some of the modules.

Too many groups

The limit is 10 including DGROUP. This message indicates that the limit was exceeded. Reduce the number of groups.

Too many libraries specified

The limit is eight libraries. This message indicates that the limit was exceeded. Reduce the number of libraries.

Too many overlays

The limit is 64 overlays. This message indicates that the limit was exceeded. Reduce the number of overlays.

Too many public symbols

The limit is 1024 public symbols. This message indicates that the limit was exceeded. Reduce the number of public symbols.

Too many segments or classes

The limit is 255 segments, classes, and groups total per module. This message indicates that the limit was exceeded. Reduce the number of segments or classes.

Unexpected end-of-file on library

Informational message. This is caused by an error in the library file. If this occurs, LINK halts.

Unexpected end-of-file on VM.TMP

Informational message. The diskette containing VM.TMP was removed. LINK halts.

Unrecognized switch

The characters specified by *z* do not uniquely identify a linker parameter (switch). See the message, "Invalid Switch."

Unresolved externals: list

The external symbols listed were not defined in the modules or library files that you specified. Do not attempt to run the executable file created by the linker. Make sure you specified all appropriate object modules and libraries. Check the source code for the program that caused the message and make corrections to that program.

VM.TMP is an illegal file name and has been ignored

VM.TMP cannot be used for an object filename. Informational message that is meant as a a warning. No action required.

Warning: no stack segment

Informational message. None of the object modules specified contain a statement allocating stack space. No action required.

SALUT Error Messages

When SALUT detects an error, a "SALUTERR" (with error message text) command line is inserted in both the formatted output and in the pre-processed output. When editing to fix the error, it is not necessary to delete this SALUTERR error line. If the .ASM file containing the SALUTERR command is passed on to the assembler, it flags that line as containing an undefined op-code, thus adding it to its count of errors.

The text of the SALUTERR statement consists of one of the following:

- Structure mismatch
- Invalid Structure statement continuation
- Invalid condition
- No Structure active
- Invalid parameter
- Required parameter missing

Terminating Errors

SALUT is written in the BASIC compiler language. There are certain errors that BASIC detects and passes on to SALUT for proper handling.

- Integer overflow
- Device not ready
- Invalid filespec
- File not found
- File already open
- File already exists
- Disk full
- Too many files or bad filespec
- Diskette write-protected
- Disk media error
- "Error";ernum;"in line";linenum

Appendix B. Appendix B. Instruction Set Summary

8088 Instruction Set Summary

Flag Registers

AF=Auxiliary carry
CF=Carry
DF=Direction
IF=Interrupt
OF=Overflow
PF=Parity
SF=Sign
TF=Trap
ZF=Zero

Legend for Flag Conditions

A=Altered to reflect results of operation
R=Replaced from storage
U=Undefined
0=Unconditionally cleared to 0
1=Unconditionally set to 1
None=No flags affected

AAA (no operands) – ASCII adjust for addition

Operands	Bytes	Example	Flags
none	1	AAA	AF=A PF=U CF=A SF=U OF=U ZF=U

AAD (no operands) – ASCII adjust for division

Operands	Bytes	Example	Flags
none	2	AAD	AF=U PF=A CF=U SF=A OF=U ZF=A

AAM (no operands) – ASCII adjust for multiply

Operands	Bytes	Example	Flags
none	1	AAM	AF=U PF=A CF=U SF=A OF=U ZF=A

AAS (no operands) – ASCII adjust for subtraction

Operands	Bytes	Example	Flags
none	1	AAS	AF=A PF=U CF=A SF=U OF=U ZF=U

ADC destination,source – Add with carry

Operands	Bytes	Example	Flags
register,register	2	ADC AX,SI	AF=A
register,memory	2-4	ADC DX,BETA [SI]	PF=U
memory,register	2-4	ADC ALPHA [BX] [SI],DI	CF=A
register,immediate	3-4	ADC BX,256	SF=U
memory,immediate	3-6	ADC GAMMA,30H	OF=U
accumulator,immediate	2-3	ADC AL,5	ZF=U

ADD destination,source – Addition

Operands	Bytes	Example	Flags
register,register	2-4	ADD CX,DX	AF=A
register,memory	2-4	ADD DI,[BX],ALPHA	PF=U
memory,register	3-4	ADD TEMP,CL	CF=A
register,immediate	3-6	ADD CL,2	SF=U
memory,immediate	2-3	ADD ALPHA,2	OF=U
accumulator,immediate		ADD AX,200	ZF=U

AND destination,source – Logical AND			
Operands	**Bytes**	**Example**	**Flags**
register,register	2	AND AL,BL	AF=U
register,memory	2-4	AND CX,FLAG WORD	PF=A
memory,register	2-4	AND ASCII [DI],AL	CF=0
register,immediate	3-4	AND CX0,F0H	SF=A
memory,immediate	3-6	AND BETA,01H	OF=0
accumulator,immediate	2-3	AND AX,01010000B	ZF=A

CALL target – Call a procedure			
Operands	**Bytes**	**Example**	**Flags**
near-proc	3	CALL NEAR_PROC	none
far-proc	5	CALL FAR_PROC	
memptr 16	2-4	CALL PROC_TABLE [SI]	
regptr 16	2	CALL AX	
memptr 32	2-4	CALL [BX.].TASK [SI]	

CBW (no operands) – Convert byte to word			
Operands	**Bytes**	**Example**	**Flags**
none	1	CBW	none

CLC (no operands) – Clear carry flag			
Operands	**Bytes**	**Example**	**Flags**
none	1	CLC	CF=0

CLD (no operands) – Clear direction flag			
Operands	**Bytes**	**Example**	**Flags**
none	1	CLD	DF=0

CLI (no operands) – Clear interrupt flag			
Operands	**Bytes**	**Example**	**Flags**
none	1	CLI	DF=0

CMC (no operands) – Complement Carry Flag			
Operands	**Bytes**	**Example**	**Flags**
none	1	CMC	CF=A

CMP destination,source – Compare destination to source

Operands	Bytes	Example	Flags
register,register	2	CMP BX,CX	AF=A
register,memory	2-4	CMP DH.ALPHA	PF=A
memory,register	2-4	CMP [BP + 2],SI	CF=A
register,immediate	3-4	CMP BL,02H	SF=A
memory,immediate	3-6	CMP [BX] RADAR [DI],3420H	OF=A
accumulator,immediate	2-3	CMP AL,00010000B	ZF=A

CMPS dest-string,source-string – Compare string

Operands	Bytes	Example	Flags
dest-string,source-string	1	CMPS BUFF1,BUFF2	AF=A PF=A CF=A SF=A OF=A ZF=A
(repeat) dest-string,source-string	1	REPE CMPS ID,KEY	

CWD (no operand) – Convert word to doubleword

Operands	Bytes	Example	Flags
none	1	CWD	none

DAA (no operand) – Decimal adjust for addition

Operands	Bytes	Example	Flags
none	1	DAA	AF=A PF=U CF=A SF=U OF=U ZF=U

DAS (no operand) – Decimal adjust for subtraction

Operands	Bytes	Example	Flags
none	1	DAS	AF=A PF=A CF=A SF=A OF=A ZF=A

DEC destination – Decrement by 1

Operands	Bytes	Example	Flags
reg16	1	DEC AX	AF=A SF=A
reg8	2	DEC AL	OF=A ZF=A
memory	2-4	DEC ARRAY [SI]	PF=A

DIV source – Division, unsigned			
Operands	**Bytes**	**Example**	**Flags**
reg8 reg16 mem8 mem16	2 2 2-4 2-4	DIV CL DIV BX DIV ALPHA DIV TABLE [SI]	AF=U PF=U CF=U SF=U OF=U ZF=U

ESC external-op code,source – Escape			
Operands	**Bytes**	**Example**	**Flags**
immediate,memory immediate,register	2-4 2	ESC 6,ARRAY [SI] ESC 20,AL	none

HLT (no operands) – Halt			
Operands	**Bytes**	**Example**	**Flags**
none	1	HLT	none

IDIV source – Integer division			
Operands	**Bytes**	**Example**	**Flags**
reg8 reg16 mem8 mem16	2 2 2-4 2-4	IDIV BL IDIV CX IDIV DIVISOR BYTE [SI] IDIV [BX],DIVISOR_WORD	AF=U PF=U CF=U SF=U OF=U ZF=U

IMUL source – Integer multiplication			
Operands	**Bytes**	**Example**	**Flags**
reg8 reg16 mem8 mem16	2 2 2-4 2-4	IMUL CL IMUL BX IMUL RATE_BYTE IMUL RATE_WORD[BP] [DI]	AF=U PF=U CF=A SF=U OF=A ZF=U

IN accumulator,port – Input byte or word			
Operands	**Bytes**	**Example**	**Flags**
accumulator,immed8 accumulator,DX	2 1	IN AL,0FFEAH IN AX,DX	none

INC destination – Increment by 1			
Operands	**Bytes**	**Example**	**Flags**
reg19 reg8 memory	1 2 2-4	INC CX INC BL INC ALPHA [DI] [BX]	AF=A SF=A OF=A ZF=A PA=A

INT interrupt-type – Interrupt

Operands	Bytes	Example	Flags
immed8(type=3) immed8(type≠3)	1 2	INT 3 INT 67	IF=0 TF=0

INTO (no operands) – Interrupt if overflow

Operands	Bytes	Example	Flags
none	1	INTO	IF=0 TF=0

IRET (no operands) – Interrupt return

Operands	Bytes	Example	Flags
none	1	IRET	AF=R PF=R CF=R SF=R DF=R TF=R IF=R ZF=R

JA/JNBE short-label – Jump if above/Jump if not below or equal

Operands	Bytes	Example	Flags
short-label	2	JA ABOVE	none

JAE/JNB short-label – Jump if above or equal/Jump if not below

Operands	Bytes	Example	Flags
short-label	2	JAE ABOVE_EQUAL	none

JB/JNAE short-label – Jump if below/Jump if not above nor equal

Operands	Bytes	Example	Flags
short-label	2	JB BELOW	none

JBE/JNA short-label – Jump if below or equal/Jump if not above

Operands	Bytes	Example	Flags
short-label	2	JNA NOT ABOVE	none

JC short-label – Jump if carry

Operands	Bytes	Example	Flags
short-label	2	JC CARRY SET	none

JCXZ short-label – Jump if CX is zero

Operands	Bytes	Example	Flags
short-label	2	JCXZ COUNT DONE	none

JE/JZ short-label – Jump if equal/Jump if zero

Operands	Bytes	Example	Flags
short-label	2	JZ ZERO	none

JG/JNLE short-label – Jump if greater/Jump if not less than

Operands	Bytes	Example	Flags
short-label	2	JG GREATER	none

JGE/JNL short-label – Jump if greater or equal/Jump if not less

Operands	Bytes	Example	Flags
short-label	2	JGE GREATER EQUAL	none

JL/JNGE short-label – Jump if less/Jump if not greater nor equal

Operands	Bytes	Example	Flags
short-label	2	JL LESS	none

JLE/JNG short-label – Jump if less or equal/Jump if not greater

Operands	Bytes	Example	Flags
short-label	2	JNG NOT GREATER	none

JMP target – Jump

Operands	Bytes	Example	Flags
short-label	2	JMP SHORT	none
near-label	3	JMP WITHIN SEGMENT	
far-label	5	JMP FAR LABEL	
memptr16	2-4	JMP [BX],TARGET	
regptr16	2	JMP CX	
memptr32	2-4	JMP OTHER SEG [SI]	

JNC short-label – Jump if not carry

Operands	Bytes	Example	Flags
short-label	2	JNC NOT CARRY	none

JNE/JNZ short-label – Jump if not equal/Jump if not zero

Operands	Bytes	Example	Flags
short-label	2	JNE NOT EQUAL	none

JNO short-label – Jump if not overflow

Operands	Bytes	Example	Flags
short-label	2	JNO NO OVERFLOW	none

JNP/JPO short-label – Jump if not parity/Jump if parity ODD

Operands	Bytes	Example	Flags
short-label	2	JPO ODD PARITY	none

JNS short-label – Jump if not sign

Operands	Bytes	Example	Flags
short-label	2	JNS POSITIVE	none

JO short-label – Jump if overflow

Operands	Bytes	Example	Flags
short-label	2	JO SIGNED_OVRFLW	none

JP/JPE short-label – Jump if parity/Jump if parity even

Operands	Bytes	Example	Flags
short-label	2	JPE EVEN_PARITY	none

JS short-label – Jump if sign

Operands	Bytes	Example	Flags
short-label	2	JS NEGATIVE	none

LAHF (no operands) – Load AH from flags

Operands	Bytes	Example	Flags
none	1	LAHF	none

LDS destination,source – Load pointer using DS

Operands	Bytes	Example	Flags
reg16,mem32	2-4	LDS SI,DATA.SEG[DI]	none

LOCK (no operands) – Lock bus

Operands	Bytes	Example	Flags
none	1	LOCK XCHG FLAG.AL	none

LODS source-string – Load string

Operands	Bytes	Example	Flags
source-string	1	LODS CUSTOMER NAME	none
(repeat) source-string	1	REP LODS NAME	

LOOP short-label – Loop

Operands	Bytes	Example	Flags
short-label	2	LOOP AGAIN	none

LOOPE/LOOPZ short-label – Loop if equal/Loop if zero

Operands	Bytes	Example	Flags
short-label	2	LOOPE AGAIN	none

LOOPNE/LOOPNZ short-label – Loop if not equal/Loop if not zero

Operands	Bytes	Example	Flags
short-label	2	LOOPNE AGAIN	none

LEA destination,source – Load effective address

Operands	Bytes	Example	Flags
reg16,mem16	2-4	LEA BX,[BP] [DI]	none

LES destination,source – Load pointer using ES

Operands	Bytes	Example	Flags
reg16,mem32	2-4	LES DI,[BX].TEXT_BUFF	none

MOV destination,source – MOVE

Operands	Bytes	Example	Flags
memory,accumulator	3	MOV ARRAY [SI],AL	none
accumulator,memory	3	MOV AX,TEMP_RESULT	
register,register	2	MOV AX,CX	
register,memory	2-4	MOV BP,STACK_TOP	
memory,register	2-4	MOV COUNT [DI],CX	
register,immediate	2-3	MOV CL,2	
memory,immediate	3-6	MOV MASK [BX][SI],2CH	
seg-reg,reg16	2	MOV ES,CX	
seg-reg,mem16	2-4	MOV DS.SEGMENT_BASE	
reg16,seg-reg	2	MOV BP,SS	
memory,seg-reg	2-4	MOV [BX].SEG_SAVE,CS	

MOVS dest-string,source-string – MOVE string

Operands	Bytes	Example	Flags
dest-string,source-string	1	MOVS LINE EDIT_DATA	none
(repeat) dest-string.source-string	1	REP MOVS SCREEN.BUFFER	

MOVSB/MOVSW (no operands) – Move string (byte/word)

Operands	Bytes	Example	Flags
none	1	MOVSB	none
(repeat) none	1	REP MOVSW	

MUL source – Multiplication, unsigned

Operands	Bytes	Example	Flags
reg8	2	MUL BL	AF=U PF=U
reg16	2	MUL CX	CF=A SF=U
mem8	2-4	MUL MONTH [SI]	OF=A ZF=U
mem16	2-4	MUL BAUD-RATE	

NEG destination – Negate

Operands	Bytes	Example	Flags
register	2	NEG AL	AF=A PF=A
memory	2-4	NEG MULTIPLIER	CF=1*
			SF=A OF=A ZF=A

NOP no operands – No operation			
Operands	**Bytes**	**Example**	**Flags**
none	1	NOP	none

NOT destination – Logical not			
Operands	**Bytes**	**Example**	**Flags**
register	2	NOT AX	none
memory	2-4	NOT CHARACTER	

OR destination,source – Logical inclusive OR			
Operands	**Bytes**	**Example**	**Flags**
register,register	2	OR AL,BL	AF=U
register,memory	2-4	OR DX,PORT ID [DI]	PF=A
memory,register	2-4	OR FLAG BYTE.CL	CF=0
accumulator,immediate	2-3	OR AL,0110110B	SF=A
register,immediate	3-4	OR SC,01FH	OF=0
memory,immediate	3-6	OR [BX] CMD WORD,0CFH	ZF=A

OUT port,accumulator – Output byte or word			
Operands	**Bytes**	**Example**	**Flags**
immed8.accumulator	2	OUT 44.AX	none
DX.accumulator	1	OUT DX.AL	

POP destination – Pop word off stack			
Operands	**Bytes**	**Example**	**Flags**
register	1	POP DX	none
seg-reg (CS illegal)	1	POP DS	
memory	2-4	POP PARAMETER	

POPF (no operands) – Pop flags off stack			
Operands	**Bytes**	**Example**	**Flags**
none	1	POPF	AF=R PF=R CF=R SF=R DF=R TF=R IF=R ZF=R

PUSH source – Push word onto stack			
Operands	**Bytes**	**Example**	**Flags**
register	1	PUSH SI	none
seg-reg (CS legal)	1	PUSH ES	
memory	2-4	PUSH PARAMETER	

PUSHF (no operands) – Push flags onto stack

Operands	Bytes	Example	Flags
none	1	PUSHF	none

RCL destination,count – Rotate left through carry

Operands	Bytes	Example	Flags
register,1	2	RCL CX,1	CF=A
register,CL	2	RCL AL,CL	OF=A
memory,1	2-4	RCL ALPHA,1	
memory,CL	2-4	RCL [BP].PARM,CL	

RCR destination,count – Rotate right through carry

Operands	Bytes	Example	Flags
register,1	2	RCR BX,1	CF=A
register,CL	2	RCR BL,CL	OF=A
memory,1	2-4	RCR [BX],STATUS,1	
memory,CL	2-4	RCR ARRAY [DI],CL	

REP (no operands) – Repeat string operation

Operands	Bytes	Example	Flags
none	1	REP MOVS DEST.SRCE	none

REPE/REPZ (no operands) – Repeat string operation equal/while zero

Operands	Bytes	Example	Flags
none	1	REPE CMPS DATA.KEY	none

REPNE/REPNZ (no operands) – Repeat string operation not equal/not zero

Operands	Bytes	Example	Flags
none	1	REPNE SCAS INPUT LINE	none

RET optional-pop-value – Return from procedure

Operands	Bytes	Example	Flags
(intra-segment,no pop)	1	RET	none
(intra-segment,pop)	3	RET 4	
(inter-segment,no pop)	1	RET	
(inter-segment,pop)	3	RET 2	

ROL destination,count – Rotate left

Operands	Bytes	Example	Flags
register,1	2	ROL BX,1	CF=A OF=A
register,CL	2	ROL DI,CL	
memory,1	2-4	ROL FLAG BYTE[DI],1	
memory,CL	2-4	ROL ALPHA,CL	

ROR destination,count

Operands	Bytes	Example	Flags
register,1	2	ROR AL,1	CF=A OF=A
register,CL	2	ROR BX,CL	
memory,1	2-4	ROR PORT STATUS,1	
memory,CL	2-4	ROR CMD WORD,CL	

SAHF (no operands) – Store AH into flags

Operands	Bytes	Example	Flags
none	1	SAHF	AF=A SF=R CF=R ZF=R PF=R

SAL/SHL destination,count – Shift arithmetic left/Shift logical left

Operands	Bytes	Example	Flags
register,1	2	SAL AL,1	CF=A OF=A
register,CL	2	SHL DI,CL	
memory,1	2-4	SHL [BX].OVERDRAW,1	
memory,CL	2-4	SAL STORE_COUNT,CL	

SAR destination,source – Shift arithmetic right

Operands	Bytes	Example	Flags
register,1	2	SAR DX,1	AF=U PF=A CF=A SF=A OF=A ZF=A
register,CL	2	SAR DI,CL	
memory,1	2-4	SAR N BLOCKS,1	
memory,CL	2-4	SAR N BLOCKS,CL	

SBB destination,source – Subtract with borrow

Operands	Bytes	Example	Flags
register,register	2	SBB BX,CX	AF=A PF=A CF=A SF=A OF=A ZF=A
register,memory	2-4	SBB DI,[BX].PAYMENT	
memory,register	2-4	SBB BALANCE,AX	
accumulator,immediate	2-3	SBB AX,2	
register,immediate	3-4	SBB CL,1	
memory,immediate	3-6	SBB COUNT[SI],10	

SCAS dest-string – Scan string

Operands	Bytes	Example	Flags
dest-string	1	SCAS INPUT_LINE	AF=A PF=A CF=A SF=A OF=A ZF=A
(repeat string)	1	REPNE SCAS BUFFER	

SHR destination,count – Shift logical right

Operands	Bytes	Example	Flags
register.1	2	SHR SI.1	CF=A OF=A
register.CL	2	SHR SI.CL	
memory.1	2-4	SHR ID BYTE [SI][BX].1	
memory.CL	2-4	SHR INPUT WORD.CL	

STC (no operands) – Set carry flag

Operands	Bytes	Example	Flags
none	1	STC	CF=1

STD (no operand) – Set direction flag

Operands	Bytes	Example	Flags
none	1	STD	DF=1

STI (no operands) – Set interrupt enable flag

Operands	Bytes	Example	Flags
none	1	STI	IF=1

STOS dest-string – Store byte or word string

Operands	Bytes	Example	Flags
dest-string	1	STOS PRINT LINE	none
(repeat) dest-string	1	REP STOS DISPLAY	

SUB destination,source – Subtraction

Operands	Bytes	Example	Flags
register,register	2	SUB XC.BX	AF=U
register,memory	2-4	SUB DX,MATH_TOTAL[SI]	PF=A
memory,register	2-4	SUB [BP+2],CL	CF=A
accumulator,immediate	2-3	SUB AL,10	SF=A
register,immediate	3-4	SUB SI,5280	OF=A
memory,immediate	3-6	SUB [BP].BALANCE,1000	ZF=A

TEST destination,source – Test or non-destructive logical AND

Operands	Bytes	Example	Flags
register,register	2	TEST SI,DI	AF=U
register,memory	2-4	TEST SI,END_COUNT	PF=A
accumulator,immediate	2-3	TEST AL,00100000B	CF=0
register,immediate	3-4	TEST BX,0CC4H	SF=A
memory,immediate	3-6	TEST RETURN CODE,01H	0F=0
			ZF=A

WAIT (no operands) – Wait while TEST pin not asserted

Operands	Bytes	Example	Flags
none	1	WAIT	none

XCHG destination,source – Exchange

Operands	Bytes	Example	Flags
accumulator,reg16	1	XCHNG AX,BX	none
memory,register	2-4	XCHG SEMAPHORE,AX	
register,register	2	XCHG AL,BL	

XLAT source-table – Translate

Operands	Bytes	Example	Flags
source-table	1	XLAT ASCII_TAB	none

XOR destination,source – Logical exclusive OR

Operands	Bytes	Example	Flags
register,register	2	XOR CX,BX	AF=U
register,memory	2-4	XOR CL,MASK BYTE	PF=A
memory,register	2-4	XOR ALPHA [SI],DX	CF=0
accumulator,immediate	2-3	XOR AL,01000010B	SF=A
register,immediate	3-4	XOR SI,00C2H	OF=0
memory,immediate	3-6	XOR RETURN CODE.0D2H	ZF=A

8087 Instruction Set Summary

In the following instruction descriptions, alternative operand forms are indicated and separated by the slash (/) notation. For example:

```
//source/destination,source
```

In this example, the operands can be written in any of three ways:

OPCODE
OPCODE *source*
OPCODE *destination,source*

```
//source
```

This example allows two operand forms:

OPCODE
OPCODE *source*

To see examples of the various forms of the floating-point instructions, use MASM to assemble the file FLINS.ASM, which is on your Macro Assembler Diskette. The results are not intended to be executed, so no object file should be produced. The command is:

```
MASM FLINS,NUL,;
```

FABS (no operands) - Absolute value

Operands	Bytes	Example	Flags
(no operands)	2	FABS	I

FADD //source/destination/source - Add real

Operands	Bytes	Example	Flags
//ST,/ST(i)/ST(i),ST	2	FADD ST,ST(4)	I,D,O, U,P
short-real	2-4	FADD AIR_TEMP [SI]	
long-real	2-4	FADD [BX].MEAN	

FADDP destination,source - Add real and pop

Operands	Bytes	Example	Flags
ST(i),ST	2	FADDP ST(2),ST	I,D,O,U,P

FBLD source - Packed decimal (BCD) load

Operands	Bytes	Example	Flags
packed-decimal	2-4	FBLD YTD_SALES	I

FBSTP destination - Packed decimal (BCD) store and pop

Operands	Bytes	Example	Flags
packed-decimal	2-4	FBSTP [BX].FORECAST	I

FCHS (no operation) - Change sign

Operands	Bytes	Example	Flags
(no operands)	2	FCHS	I

FCLEX/FNCLEX (no operands) - Clear exceptions

Operands	Bytes	Example	Flags
(no operands)	2	FNCLEX	none

FCOM //source - Compare real

Operands	Bytes	Example	Flags
//ST(i)	2	FCOM ST(1)	I,D
short-real	2-4	FCOM [BP].UPPER_LIMIT	
long-real	2-4	FCOM WAVELENGTH	

FCOMP //source - Compare real and pop

Operands	Bytes	Example	Flags
//ST(i)	2	FCOMP ST(2)	I,D
short-real	2-4	FCOMP [BP+2].N_READINGS	
long-real	2-4	FCOMP DENSITY	

FCOMPP (no operands) - Compare real and pop twice

Operands	Bytes	Example	Flags
(no operands)	2	FCOMPP	I,D

FDECSTP (no operands) - Decrement stack pointer

Operands	Bytes	Example	Flags
(no operands)	2	FDECSTP	None

FDISI/FNDISI (no operands) - Disable interrupts

Operands	Bytes	Example	Flags
(no operands)	2	FDISI	None

FDIV //source/destination,source - Divide real

Operands	Bytes	Example	Flags
//ST(i),ST	2	FDIV	I,D,O,U,P
short-real	2-4	FDIV DISTANCE	
long-real	2-4	FDIV ARC[DI]	

FDIVP destination,source - Divide real and pop

Operands	Bytes	Example	Flags
ST(i),ST	2	FDIVP ST(4),ST	I,D,Z,O,U,P

FDIVR //source/destination,source - Divide real reversed

Operands	Bytes	Example	Flags
//ST,ST(i)/ST(i),ST	2	FDIVR ST(2),ST	I,D,Z,O,U,P
short-real	2-4	FDIVR [BX].PULSE_RATE	
long-real	2-4	FDIVR RCRDR.FRQNCY	

FDIVRP destination,source - Divide real reversed and pop

Operands	Bytes	Example	Flags
ST(i),ST	2	FDIVRP ST(i),ST	I,D,Z,O,U,P

FENI/FNENI (no operands) - Enable interrupts

Operands	Bytes	Example	Flags
(no operands)	2	FNENI	none

FFREE destination - Free register

Operands	Bytes	Example	Flags
ST(i)	2	FFREE ST(i)	none

FIADD source - Integer add

Operands	Bytes	Example	Flags
word-integer	2-4	FIADD-DISTANCE_TRAVELED	I,D,
short-integer	2-4	FIADD PULSE_COUNT [SI]	O,P

FICOM source - Integer compare

Operands	Bytes	Example	Flags
word-integer	2-4	FICOM TOOL.N_PASSES	I,D
short-integer	2-4	FICOM [BP+4].PARM_COUNT	

FICOMP source - Integer compare and pop

Operands	Bytes	Example	Flags
word-integer	2-4	FICOMP [BP].LIMIT [SI]	I,D
short-integer	2-4	FICOMP N_SAMPLES	

FIDIV source - Integer divide

Operands	Bytes	Example	Flags
word-integer	2-4	FIDIV SURVEY.OBSERVATIONS	I,D,Z,
short-integer	2-4	FIDIV RELATIVE_ANGLE [DI]	O,U,P

FIDIVR source - Integer divide reversed

Operands	Bytes	Example	Flags
word-integer	2-4	FIDIVR [BP].X_COORD	I,D,Z,
short-integer	2-4	FIDIVR FREQUENCY	O,U,P

FILD source - Integer load

Operands	Bytes	Example	Flags
word-integer	2-4	FILD [BX].SEQUENCE	I
short-integer	2-4	FILD STANDOFF [DI]	
long-integer	2-4	FILD RESPONSE.COUNT	

FIMUL source - Integer multiply

Operands	Bytes	Example	Flags
word-integer	2-4	FIMUL BEARING	I,D, O,P
short-integer	2-4	FIMUL POSITION.Z_AXIS	

FINCSTP (no operands) - Increment stack pointer

Operands	Bytes	Example	Flags
(no operands)	2	FINCSTP	None

FINIT/FNINIT (no operands) - Initialize processor

Operands	Bytes	Example	Flags
(no operands)	2	FINIT	None

FIST (no operands) - Integer store

Operands	Bytes	Example	Flags
word-integer	2-4	FIST OBS.COUNT[SI]	I,P
short-integer	2-4	FIST [BP].FACTORED_PULSES	

FISTP destination - Integer store and pop

Operands	Bytes	Example	Flags
word-integer	2-4	FISTP [BX].ALPHA_COUNT [SI]	I,P
short-integer	2-4	FISTP CORRECTED_TIME	
long-integer	2-4	FISTP PANEL.N_READINGS	

FISUB source - Integer subtract

Operands	Bytes	Example	Flags
word-integer	2-4	FISUB BASE_FREQUENCY	I,D, O,P
short-integer	2-4	FISUB TRAIN_SIZE [DI]	

FISUBR source - Integer subtract reversed

Operands	Bytes	Example	Flags
word-integer	2-4	FISUBR FLOOR [BX] [SI]	I,D,
short-integer	2-4	FISUBR BALANCE	O,P

FLD source - Load real

Operands	Bytes	Example	Flags
ST(i)	2	FLD ST(O)	I,D
short-real	2-4	FLD READING [SI].PRESSURE	
long-real	2-4	FLD [BP].TEMPERATURE	
temp-real	2-4	FLD SAVEREADING	

FLDCW source - Load control word

Operands	Bytes	Example	Flags
2-bytes	2-4	FLDCW CONTROL_WORD	None

FLDENV source - Load environment

Operands	Bytes	Example	Flags
14-bytes	2-4	FLDENV [BP+6]	None

FLDLG2 (no operands) - Load log base 10 of 2

Operands	Bytes	Example	Flags
(no operands)	2	FLDLG2	I

FLDLN2 (no operands) - Load log base e of 2

Operands	Bytes	Example	Flags
(no operands)	2	FLDLN2	I

FLDL2E (no operands) - Load log base 2 of e

Operands	Bytes	Example	Flags
(no operands)	2	FLDL2E	I

FLDL2T (no operands) - Load log base 2 of 10

Operands	Bytes	Example	Flags
(no operands)	2	FLDL2T	I

FLDPI (no operands) - Load PI

Operands	Bytes	Example	Flags
(no operands)	2	FLDPI	I

FLDZ (no operands) - Load +0.0

Operands	Bytes	Example	Flags
(no operands)	2	FLDZ	I

FLD1 (no operands) - Load +1.0

Operands	Bytes	Example	Flags
(no operands)	2	FLD1	I

FMUL //source/destination,source - Multiply real

Operands	Bytes	Example	Flags
//ST(i),ST/ST,ST(i)[1]	2	FMUL ST,ST(3)	I,D,O, U,P
//ST(i),ST/ST,ST(i)	2	FMUL ST,ST(3)	
short-real	2-4	FMUL SPEED_FACTOR	
long-real	2-4	FMUL [BP].HEIGHT	
long-real	2-4	FMUL [BP].HEIGHT	

[1] occurs when one or both operands is "short" —it has 40 trailing zeros in its fraction (that is, it was loaded from a short-real memory operand).

FMULP destination,source - Multiply real and pop

Operands	Bytes	Example	Flags
ST(i),ST[1]	2	FMULP ST(1),ST	I,D,O, U,P
ST(i),ST	2	FMULP ST(1),ST	

[1] occurs when one or both operands is "short" —it has 40 trailing zeros in its fraction (that is, it was loaded from a short-real memory operand).

FNOP (no operands) - No operation

Operands	Bytes	Example	Flags
(no operands)	2	FNOP	None

FPATAN (no operands) - Partial arc-tangent

Operands	Bytes	Example	Flags
(no operands)	2	FPATAN	U,P (operands not checked)

FPREM (no operands) - Partial remainder

Operands	Bytes	Example	Flags
(no operands)	2	FPREM	I,D,U

FPTAN (no operands) - Partial tangent

Operands	Bytes	Example	Flags
(no operands)	2	FPTAN	I,P (operands not checked)

FRNDINT (no operands) - Round to integer

Operands	Bytes	Example	Flags
(no operands)	2	FRNDINT	I,P

FRSTOR source - Restore saved state

Operands	Bytes	Example	Flags
94-bytes	2-4	FRSTOR [BP]	None

FSAVE/FNSAVE destination - Save state

Operands	Bytes	Example	Flags
94-bytes	2-4	FSAVE [BP]	None

FSCALE (no operands) - Scale

Operands	Bytes	Example	Flags
(no operands)	2	FSCALE	I,O,U

FSQRT (no operands) - Square root

Operands	Bytes	Example	Flags
(no operands)	2	FSQRT	I,D,P

FST destination - Store real

Operands	Bytes	Example	Flags
ST(i)	2	FST ST(3)	I,O, U,P
short-real	2-4	FST CORRELATION [DI]	
long-real	2-4	FST MEAN_READING	

FSTCW/FNSTCW destination - Store control word

Operands	Bytes	Example	Flags
2-bytes	2-4	FSTCW SAVE_CONTROL	None

FSTENV/FNSTENV destination - Store environment

Operands	Bytes	Example	Flags
14-bytes	2-4	FSTENV [BP]	None

FSTP destination - Store real and pop

Operands	Bytes	Example	Flags
ST(i)	2	FSTP ST(2)	I,O, U,P
short-real	2-4	FSTP [BX].ADJUSTED_RPM	
long-real	2-4	FSTP TOTAL_DOSAGE	
temp-real	2-4	FSTP REG_SAVE [SI]	

FSTSW/FNSTSW destination - Store status word

Operands	Bytes	Example	Flags
2-bytes	2-4	FSTSW SAVE_STATUS	None

FSUB //source/destination,source - Subtract real

Operands	Bytes	Example	Flags
//ST,ST(i)/ST(i),ST	2	FSUB ST,ST(2)	I,D,O, U,P
short-real	2-4	FSUB BASE_VALUE	
long-real	2-4	FSUB COORDINATE.X	

FSUBP destination,source - Subtract real and pop

Operands	Bytes	Example	Flags
ST(i),ST	2	FSUBP ST(2),ST	I,D,O,U,P

FSUBR //source/destination,source - Subtract real reversed

Operands	Bytes	Example	Flags
//ST,ST(i)/ST(i),ST	2	FSUBR ST,ST(1)	I,D,O, U,P
short-real	2-4	FSUBR VECTOR[SI]	
long-real	2-4	FSUBR [BX].INDEX	

FSUBRP destination,source - Subtract real reversed and pop

Operands	Bytes	Example	Flags
ST(i),ST	2	FSUBRP ST(1),ST	I,D,O,U,P

FTST (no operands) - Test stack top against +0.0

Operands	Bytes	Example	Flags
(no operands)	2	FTST	I,D

FWAIT (no operands) - (CPU) Wait while 8087 is busy

Operands	Bytes	Example	Flags
(no operands)	1	FWAIT	None (CPU instruction)

FXAM (no operands) - Examine stack top

Operands	Bytes	Example	Flags
(no operands)	2	FXAM	None

FXCH //destination - Exchange registers

Operands	Bytes	Example	Flags
//ST(i)	2	FXCH ST(2)	I

FXTRACT (no operands) - Extract exponent and significand

Operands	Bytes	Example	Flags
(no operands)	2	FXTRACT	I

FYL2X (no operands) - Y • Log base 2 of X

Operands	Bytes	Example	Flags
(no operands)	2	FYL2X	P (operands not checked)

FYL2XP1 (no operands) - Y • log base 2 of (X+1)

Operands	Bytes	Example	Flags
(no operands)	2	FYL2XP1	P (operands not checked)

F2XM1 (no operands) - 2 to the x power -1			
Operands	**Bytes**	**Example**	**Flags**
(no operands)	2	F2XM1	U,P (operands not checked)

80286 Instruction Set Summary

Flag Registers

AF=Auxiliary carry
CF=Carry
DF=Direction
IF=Interrupt
OF=Overflow
PF=Parity
SF=Sign
TF=Trap
ZF=Zero

Legend for Flag Conditions

A=Altered to reflect results of operation
R=Replaced from storage
U=Undefined
0=Unconditionally cleared to 0
1=Unconditionally set to 1
None=No flags affected

BOUND destination,source – Detect value out of range

Operands	Bytes	Example	Flags
register,memory	2	BOUND AX,ALPHA	none

ENTER immediate-word,immediate-byte – Procedure entry

Operands	Bytes	Example	Flags
Locals,level	4	ENTER 28,3	none

IMUL destination,source,immediate – Integer multiplication

Operands	Bytes	Example	Flags
immed8	3	IMUL 6	AF=U PF=U
immed16	4	IMUL 20	CF=A SF=U
reg8	2	IMUL CL	OF=A ZF=U
reg16	2	IMUL BX	
mem8	2-4	IMUL RATE_BYTE	
mem16	2-4	IMUL RATE_WORD[BP] [DI]	

INS source-string,port – Input string

Operands	Bytes	Example	Flags
dest-string,port	1	INS BUFF1,0FFEAH	none

LEAVE (no operands) – Procedure exit

Operands	Bytes	Example	Flags
none	1	LEAVE	none

OUTS port,source-string – Output string

Operands	Bytes	Example	Flags
port,source-string	1	OUTS 44.BUFF2	none

POPA (no operands) – Pop all registers

Operands	Bytes	Example	Flags
none	1	POPA	none

PUSH source – Push immediate

Operands	Bytes	Example	Flags
immed8	2	PUSH RATE_BYTE	none
immed16	3	PUSH RATE_WORD	

PUSHA (no operands) – Push all registers

Operands	Bytes	Example	Flags
none	1	PUSHA	none

Appendix C. Pseudo Operations Summary

This appendix lists the pseudo-ops in their related groups: Conditional, Data, Listing, and Macro. The name and format for each pseudo-op is defined.

Type	Pseudo-ops		
Conditional	ELSE	IFDEF	IFNB
	ENDIF	IFDIF	IFNDEF
	IF	IFE	IF1
	IFB	IFIDN	IF2
Data	ASSUME	ENDS	NAME
	COMMENT	EQU	ORG
	DB	= (Equal Sign)	PROC
	DD		PUBLIC
	DQ	EVEN	.RADIX
	DT	EXTRN	RECORD
	DW	GROUP	SEGMENT
	END	INCLUDE	STRUC
	ENDP	LABEL	
Listing	.CREF	PAGE	TITLE
	.LALL	.SALL	.XALL
	.LFCOND	.SFCOND	.XCREF
	.LIST	SUBTTL	.XLIST
	%OUT	.TFCOND	
Macro	ENDM	IRPC	PURGE
	EXITM	LOCAL	REPT
	IRP	MACRO	
Mode	.286C	.8086	.8087

Conditional Pseudo Operations

[ELSE]

IF *expression*

IFB <*operand*>

IFDEF *symbol*

IFDIF <*operand-1*>,<*operand-2*>

IFE *expression*

IFIDN <*operand-1*>,<*operand-2*>

IFNB <*operand*>

IFNDEF *symbol*

IF1 (*no operand*)

IF2 (*no operand*)

ENDIF (*no operand*)

Data Pseudo Operations

ASSUME *seg-reg*:*seg-name*[,...]
OR
ASSUME *seg-reg*:NOTHING[,...]
OR
ASSUME NOTHING

COMMENT *delimiter text delimiter*

variable-name DB *expression*

variable-name DW *expression*

variable-name DD *expression*

variable-name DQ *expression*

variable-name DT *expression*

END *expression*

name EQU *expression*

label= *expression*

EVEN (no operand)

EXTRN *name*:*type*[,...]

name GROUP *seg-name* [,...]

name LABEL *type*

NAME *module-name*

ORG *expression*

procedure-name PROC [NEAR]
OR
procedure-name PROC [FAR]

PUBLIC *symbol*[,...]

PURGE *symbol*

.RADIX *expression*

recordname RECORD *fieldname*:*width*[=*exp*][,...]

segname SEGMENT [*align*][*combine*]
['*class*']
•
•
•
segname ENDS

structurename STRUC

•
•
•
structurename ENDS

Listing Pseudo Operations

.CREF (no operand)

.LALL (no operand)

.LFCOND (no operand)

.LIST (no operand)

%OUT *text*

PAGE [*expression*]

.SALL (no operand)

.SFCOND (no operand)

SUBTTL *text*

.TFCOND (no operand)

TITLE *text*

.XALL (no operand)

.XCREF (no operand)

.XLIST (no operand)

Macro Pseudo Operations

ENDM (no operand)

EXITM (no operand)

IRP *dummy,<operandlist>* • • • ENDM

IRPC *dummy,string* or (<*string*>) • • • ENDM

name MACRO *dummylist* • • • ENDM

name parmlist

LOCAL *dummylist*

REPT *expression* • • ENDM

Special Macro operators

& ;; ! %

Mode Pseudo Operations

.286C (no operand)

.8086 (no operand)

.8087 (no operand)

Appendix D. The Sample Sessions

Introduction

The following sample session routines are highly tutorial. The source files are designed to demonstrate important coding and syntax conventions as well as provide examples of entering data from the keyboard and displaying data on the monitor. The sample sessions demonstrate the small assembler (ASM), macro assembler (MASM), and the utilities included with this product.

The samples are:

- The TRYCOM sample session using:
  - The small assembler (ASM)
  - The macro assembler (MASM)
- The SALUT sample session
- Assembler source coding aids

The TRYCOM small assembler sample session provides the source files and the command sequence to build an assembler diskette and execute the steps of the development process.

The development process includes the steps necessary to build an executable program. In the assembler, that process starts at the *source statement* level in the source file and ends with an executable program. See Chapter 1 for information and a flowchart on the development process.

The TRYCOM macro assembler sample session performs similar steps, using a batch file, to build an assembler diskette and issue the development process command sequence. This session illustrates the use of a batch file while demonstrating the development process. The TRYCOM macro assembler sample session source files (TRYCOM1M.ASM and TRYCOM2M.ASM) include macro statements requiring the use of the macro assembler (MASM) for the development process.

The SALUT sample session is a comprehensive session:

- Creating and using batch files.
- Demonstrating how the SALUT utility formats a source file.
- Using the SALUT, MASM, LIB, and LINK utilities in the development process.

The assembler source coding aids include:

- Source files providing the framework within which source code can be inserted.
- An 8087 example source file.

Requirements

- At least one diskette drive (two preferred).
- The IBM Personal Computer *Disk Operating System (DOS).*
- Three blank diskettes.
- A printer (optional, but desirable).
- A copy of the Macro Assembler Diskette.
- The small assembler (ASM) requires at least 96K memory.
- The macro assembler (MASM) requires at least 128K memory.

TRYCOM Sample Sessions

The two sets of source files (modules) used in this sample session are:

- TRYCOM1S.ASM and TRYCOM2S.ASM (for the small assembler)
- TRYCOM1M.ASM and TRYCOM2M.ASM (for the macro assembler)

Study the comments on each line of source using an editor (such as, DOS EDLIN or The IBM Personal Computer *Personal Editor*) or the DOS TYPE command (if you have a printer, press the Ctrl-PrtSc keys first for a printed copy). The comments explain the purpose of each assembler statement.

The TRYCOM examples demonstrate the following:

- Structure of a .COM style source file

  Note: See Chapter 3 for information on the structure of .COM and .EXE files.

- Linking of two modules

- The use of EXE2BIN to convert an .EXE file into an executable .COM file.

See Chapter 1 for information about:

- The Macro Assembler diskette.
- Backing up the Macro Assembler diskette.
- The application development process.
- Files and filenames.

Small Assembler Session

Small Assembler Samples Diskette

Do the following steps to create your Small Assembler Samples diskette.

1. If you have not done so, back up your Macro Assembler diskette using the procedure in Chapter 1.

2. Format a blank diskette and label it *Small Assembler Samples.*

3. Copy the following files from your DOS diskette onto the Small Assembler Samples diskette:

   - COMMAND.COM
   - EXE2BIN.EXE

   And for DOS 1.1 users only:

   - LINK.EXE

4. Copy the following files from your Macro Assembler diskette onto the Small Assembler Samples diskette:

   - ASM.EXE
   - CREF.EXE
   - TRYCOM1S.ASM
   - TRYCOM2S.ASM

   And for DOS 2.0 (or later) users only:

   - LINK.EXE

Small Assembler Sample Procedure

Do the following development process steps:

1. Place your DOS diskette in drive A and perform a system reset (Ctrl-Alt-Del keys together). Enter the date and time. The system reset is necessary to ensure the compatibility of this procedure with various machine configurations. Although it may seem unnecessary, please do so.

2. Place your Small Assembler Samples diskette in drive A and enter the following commands at the DOS prompt:

   ```
   A:
   ASM TRYCOM1S,,,;
   ```

   On completion enter:

   ```
   ASM TRYCOM2S,,,;
   ```

   On completion enter:

   ```
   DIR TRYCOM*
   ```

   Observe from the directory that the small assembler (ASM) created the following files:

   - TRYCOM1S.OBJ
   - TRYCOM1S.LST
   - TRYCOM1S.CRF
   - TRYCOM2S.OBJ
   - TRYCOM2S.LST
   - TRYCOM2S.CRF

3. Create the cross reference files with the following commands:

   ```
   CREF TRYCOM1S;
   CREF TRYCOM2S;
   ```

   On completion enter:

   ```
   DIR TRYCOM*
   ```

   Observe from the directory that the cross reference utility (CREF) created the following files:

   - TRYCOM1S.REF
   - TRYCOM2S.REF

   After running CREF, the temporary TRYCOM1S.CRF and TRYCOM2S.CRF files (created by the assembler) can be deleted:

   ```
   DEL *.CRF
   ```

4. LINK TRYCOM1S.OBJ with TRYCOM2S.OBJ using the following command:

   ```
   LINK TRYCOM1S+TRYCOM2S,TRYCOMS,TRYCOMS/M;
   ```

   The LINK message:

   ```
   Warning: No STACK segment
   There was 1 error detected.
   ```

   is expected because you are building a .COM style program. Enter:

   ```
   DIR TRYCOM*
   ```

Observe from the directory that the Linker Utility (LINK) created the following files:

- TRYCOMS.EXE
- TRYCOMS.MAP

After running LINK, the temporary TRYCOM1S.OBJ and TRYCOM2S.OBJ files (created by the assembler) can be deleted:

```
DEL *.OBJ
```

5. Use the EXE2BIN utility to convert the TRYCOMS.EXE file to a .COM file with the following command:

```
EXE2BIN TRYCOMS TRYCOMS.COM
```

On completion enter:

```
DIR TRYCOM*
```

Observe from the directory that the EXE2BIN Utility created the following file:

- TRYCOMS.COM

After running EXE2BIN, the temporary TRYCOMS.EXE file (created by LINK) can be deleted:

```
DEL TRYCOMS.EXE
```

6. Execute the completed TRYCOMS.COM program with the following command:

```
TRYCOMS
```

7. Print (if you have a printer) or display each of the following sample files:

    - TRYCOM1S.LST (produced by ASM)
    - TRYCOM2S.LST (produced by ASM)
    - TRYCOM1S.REF (produced by CREF)
    - TRYCOM2S.REF (produced by CREF)
    - TRYCOMS.MAP (produced by LINK)

Small Assembler Session Summary

This sample process:

- Produced a small assembler sample diskette.

- Demonstrated a sample of commands to use the ASM, CREF, LINK, and EXE2BIN utilities.

- Produced two assembler listing files.

- Produced two cross reference files.

- Produced a link map file.

- Produced a .COM style program.

The small assembler sample session command sequence was chosen to demonstrate some of the utilities included on the Macro Assembler diskette and produce sample files for your reference.

Macro Assembler Session

This sample session demonstrates the use of the macro assembler (MASM) to assemble source files containing macros. A batch file containing the command sequence demonstrates an alternate method of performing the development process.

> **Note:** The Macro Assembler (MASM) requires at least 128K memory.

Macro Assembler Samples Diskette

Do the following steps to create your Macro Assembler Samples diskette.

1. Format a blank diskette and label it *Macro Assembler Samples.*

2. Copy the following batch file from your Macro Assembler diskette onto the Macro Assembler Samples diskette:

   SAMPLE2.BAT

Macro Assembler Sample Procedure

Do the following development process steps:

1. Place your DOS diskette in drive A and perform a system reset (Ctrl-Alt-Del keys together). Enter the date and time. The system reset is necessary to ensure the compatibility of this procedure with various machine configurations. Although it may seem unnecessary, please do so.

2. Place your Macro Assembler Samples diskette in drive A and enter the following commands at the DOS prompt:

```
A:
SAMPLE2
```

3. Observe and respond to the commands and their results on your screen. Note how this session is doing the same command sequence as the previous small assembler sample session, with these exceptions:

   - Batch file operation instead of DOS prompt responses.
   - The names of the source and output files.
   - Use of MASM to assemble the source files (required because the source files contain macros).

   Because of the similarities you may want to follow the small assembler sample session in the previous section for additional information about each step.

See Chapters 4 through 8 for information on each of the utilities and their various command options.

- Chapter 4, "Structured Assembly Language Utilities (SALUT)"
- Chapter 5, "Macro Assembler (ASM and MASM)"
- Chapter 6, "Cross Reference Utility (CREF)"
- Chapter 7, "The Library Manager (LIB)"
- Chapter 8, "The Linker Utility (LINK)"

SALUT Sample Session

The source files (modules) used in this sample session are:

- STACK.ASM (assembler source)
- SALEXINC.ASM (included source)
- SALEX.SAL (SALUT source)
- SALEX1.SAL (SALUT source)
- SALEX2.SAL (SALUT source)
- SALEX3.SAL (SALUT source)
- SALEX4.SAL (SALUT source)

Of the seven source files, one file, STACK.ASM is processed by the assembler. One source file, SALEXINC.ASM, is an include file referenced by the SALEX?.SAL source files. The remaining five source files, SALEX.SAL, SALEX1.SAL, SALEX2.SAL, SALEX3.SAL, and SALEX4.SAL, are SALUT source files (denoted by the filename extension .SAL) and are pre-processed by SALUT to produce assembler source files (denoted by the filename extension .ASM).

The SALEX files have a .SAL filename extension. The .SAL filename extension identifies these files as Structured assembly language which must be processed into .ASM files by SALUT before they can be assembled.

Study the comments on each line of source using an editor (such as, DOS EDLIN or The IBM Personal Computer *Personal Editor*) or the DOS TYPE command (if you have a printer, press the Ctrl-PrtSc keys first for a printed copy). The comments explain the purpose of each assembler statement.

The SALUT example source files demonstrate the following:

- Structure of an .EXE source file

  Note: See Chapter 3 for information on the structure of .COM and .EXE files.

- Stacks (user defined)
- Including a source file into another source file

  Note: See the "Include Pseudo-op" in Chapters 1 and 2 of the *Macro Assembler Reference* book.

- Batch file processing
- Formatting of source files
- SALUT preprocessing
- Using the Library Manager
- Linking of two (or more) modules

SALUT Samples Diskette

Do the following steps to create your SALUT Samples diskette:

1. Format a blank diskette and label it *SALUT Samples.*

2. Copy the following files from your DOS diskette onto the SALUT Samples diskette:

   - COMMAND.COM

   And for DOS 1.1 users only:

   - LINK.EXE

3. Copy the following files from your Macro Assembler diskette onto the SALUT Samples diskette:

   - LIB.EXE
   - MASM.EXE
   - SALUT.EXE
   - STACK.ASM
   - SALEXINC.ASM
   - SALEX.SAL
   - SALEX1.SAL
   - SALEX2.SAL
   - SALEX3.SAL
   - SALEX4.SAL

   And for DOS 2.0 (or later) users only:

   - LINK.EXE

4. Create the following batch file on the SALUT Samples diskette using an editor (such as, DOS EDLIN or The IBM Personal Computer *Personal Editor*):

   - Filename and extension of file:

     ```
     SA.BAT
     ```

   - Contents of file:

     ```
     SALUT %1
     MASM %1,,;
     DEL %1.ASM
     ```

     Note: See "batch files" in your DOS book for more information on creating and using batch files.

5. Create the following batch file on the SALUT Samples diskette:

   - Filename and extension of file:

     ```
     L.BAT
     ```

   - Contents of file:

     ```
     LIB SAL.LIB-+%1,SALLIB.LST;
     DEL %1.OBJ
     ```

6. Your SALUT Samples diskette now contains the following files:

```
COMMAND  COM
LIB      EXE
MASM     EXE
SALUT    EXE
STACK    ASM
SALEXINC ASM
SALEX    SAL
SALEX1   SAL
SALEX2   SAL
SALEX3   SAL
SALEX4   SAL
LINK     EXE
SA       BAT
L        BAT
```

If not, review the previous procedure and correct before continuing.

SALUT Sample Procedure

Do the following development process steps:

1. Place your DOS diskette in drive A and perform a system reset (Ctrl-Alt-Del keys together). Enter the date and time. The system reset is necessary to ensure the compatibility of this procedure with various machine configurations. Although it may seem unnecessary, please do so.

2. Place your SALUT Samples diskette in drive A and enter the following command (set default drive to A) at the DOS prompt:

```
A:
```

3. Assemble STACK.ASM with the following command:

```
MASM STACK,,;
```

4. Using the following DOS TYPE command, look at the SALEX3.ASM file. Note how the statement fields (name, operation, operand, comment) do not align (the code looks busy and confusing).

```
TYPE SALEX3.SAL
```

> **Note:** Use the Ctrl-Num Lock keys to stop scrolling (any key starts the scrolling again) as you view the file.

5. Enter the following commands to call your SA.BAT batch file, substitute the specified *filename*, and perform the operations of the batch file:

```
SA SALEX
SA SALEX1
SA SALEX2
SA SALEX3
SA SALEX4
```

> **Note:** Do not specify the filename extension in the above commands. SALUT expects a .SAL extension and the assembler expects an .ASM extension.

The batch file performed the following sequence on each *filename* file:

- The SALUT utility:
  - Renamed the original *filename*.SAL file *filename*.BAK.
  - Loaded *filename*.BAK, formatted the file and copied it back to the diskette as *filename*.SAL.
  - Processed the SALUT statements and generated the *filename*.ASM assembler source file.
- The macro assembler (MASM) assembled *filename*.ASM (including SALEXINC.ASM) creating the following files.
  - *filename*.LST.
  - *filename*.OBJ.
- Deleted the temporary *filename*.ASM file (created by SALUT).

Enter the following directory command to see the results of the previous operations:

```
DIR S*.*
```

6. Early in this procedure you looked at the SALEX3.SAL file and noted the poor alignment of statements. Repeat the same procedure (using the following DOS TYPE command), look at the SALEX3.SAL file. Note how the statement fields (name, operation, operand, comment) have been aligned properly by SALUT.

```
TYPE SALEX3.SAL
```

> **Note:** The original poorly aligned file is now SALEX3.BAK. Be aware that, SALUT will overlay SALEX3.BAK, if this session is repeated, with the properly formatted SALEX3.SAL.

7. Enter the following commands to call your L.BAT batch file, substitute the specified *filename*, and perform the operations of the batch file.

   Because this batch file will delete and then add the object file to the library, you can expect a warning that the library cannot delete a member that is not there. Ignore the warning.

   ```
   L STACK
   L SALEX1
   L SALEX2
   L SALEX3
   L SALEX4
   ```

   > **Note:** Do not specify the filename extension in the above commands. The Library Manager expects a .OBJ extension.

   The batch file performed the following sequence on each *filename* file:

   - The Library Manager (LIB) utility:
     - Creates a library file called SAL.LIB if it does not already exist.
     - Creates a library listing file called SALLIB.LST if it does not already exist.
     - Deletes the previous *filename*.OBJ file from the SAL.LIB library and adds the *filename*.OBJ file to the SAL.LIB library.
     - Updates the library listing file.
   - Deleted the temporary *filename*.OBJ file (created by the macro assembler).

Enter the following directory command to see the results of the previous operations:

```
DIR S*.*
```

Look at the members of the new library by entering:

```
TYPE SALLIB.LST
```

8. Enter the following command:

   ```
   LINK SALEX,,,SAL.LIB/M;
   ```

   The Linker (LINK) links the main module, SALEX.OBJ, to the SALEX Library members forming an executable file, SALEX.EXE. You do not have to provide the .OBJ filename extension because the Linker automatically looks for this extension. The Linker also produces a SALEX.MAP file. Look at the .MAP file by using the DOS TYPE command.

9. Execute the application by entering only the application name, SALEX:

   ```
   SALEX
   ```

10. Print (if you have a printer) or display each of the following sample files:

    - STACK.ASM (sample source)
    - SALEXINC.ASM (sample source)
    - SALEX.SAL (formatted by SALUT)
    - SALEX1.SAL (formatted by SALUT)
    - SALEX2.SAL (formatted by SALUT)
    - SALEX3.SAL (formatted by SALUT)
    - SALEX4.SAL (formatted by SALUT)
    - SALEX3.BAK (sample source)
    - STACK.LST (produced by MASM)
    - SALEX.LST (produced by MASM)
    - SALEX1.LST (produced by MASM)
    - SALEX2.LST (produced by MASM)
    - SALEX3.LST (produced by MASM)
    - SALEX4.LST (produced by MASM)
    - SALLIB.LST (produced by LIB)
    - SALEX.MAP (produced by LINK)

SALUT Sample Session Summary

This sample process:

- Produced a SALUT sample diskette.
- Demonstrated a sample batch file.
- Demonstrated a sample of commands to use the SALUT MASM, LIB, and LINK, utilities.
- Produced formatted sample source files.
- Produced sample macro assembler (MASM) listing files.
- Produced a sample object library file and listing.
- Produced a sample link map file.
- Produced a .EXE style program.

The SALUT sample session command sequence was chosen to demonstrate some of the utilities included on the Macro Assembler diskette and produce sample files for your reference.

Assembler Source Coding Aids

Skeletal Assembler Source

Three skeletal assembler source files, included on the Macro Assembler diskette, provide the framework within which source code can be inserted:

- SKELCOM.ASM
- SKELEXE.ASM
- SKELEXE2.ASM

The SKELCOM.ASM sample source file is .COM structured. The SKELEXE.ASM and SKELEXE2.ASM sample source files are .EXE structured.

When creating a .COM style application, copy the *skeleton* file, SKELCOM.ASM, to the desired source name, and use an editor to insert your application code into this framework. When creating a .EXE style application, use the SKELEXE.ASM file.

The SKELEXE2.ASM file is designed specifically for DOS version 2.0 (or later), and is the recommended version for applications that are only intended to run on DOS 2.0 (or later). The DOS linkage for SKELEXE2.ASM is not compatible with DOS 1.1, although the SKELEXE.ASM version will run on either DOS level.

8087 Assembler Sample Source.

An 8087 sample source file, included on the Macro Assembler diskette, provides examples of 8087 floating-point instructions:

- FLINS.ASM

Assemble the FLINS.ASM sample source file using the macro assembler (MASM), and print or display the listing file FLINS.LST. This is not an executable program and is only intended to provide examples of 8087 instructions. The listing can be used as a 8087 floating-point reference.

Glossary

allocate. To assign a resource for use in carrying out a specific task.

ASCII. American National Standard Code for Information Interchange. The standard code, using a coded character set consisting of 7-bit coded characters (8 bits including the parity bit). The ASCII set consists of control and graphic characters.

.ASM. A filename extension that identifies the source file for the assembler.

backup. Pertaining to copying your master diskettes or files to formatted diskettes for use in case of a malfunction or loss of data.

batch file. A batch file is a file containing one or more commands that DOS executes one at a time. All batch files must have a file name extension of .BAT.

bit. A binary digit. One bit is the smallest unit of memory in a computer.

byte. Eight bits; enough storage for one character.

call. To bring a computer program or procedure into effect, usually by specifying the entry conditions and moving to an entry point.

code. A set of rules specifying the manner in which data can be represented in a discrete form. A set of instructions specifying the manner in which data is to be manipulated.

.COM file. A .COM file is the output file produced by the EXE2BIN utility. A .COM file is an executable program.

comment. A statement or portion of a statement used to document a program. Comments include information helpful in reviewing the output listing.

CON. The device identification for the operators console. Output to CON goes to the monitor (screen); input from CON is read from the keyboard. Used in the listing field of assembler and CREF commands.

concatenate. To physically place (link) together separate items.

constant. A data item of non-varying fixed value.

control block. A storage area used by a computer program to hold control information.

CREF. Cross Reference Facility (utility).

Lists the cross references for symbols and locations of all intermodule references.

.CRF. A filename extension that identifies the cross reference file to be passed to the CREF utility.

data item. Any constant or variable value.

debug. To detect, trace, and correct errors in a computer program.

DEBUG. A DOS utility used in the debug process.

default. A value, attribute, or option assumed when none has been specified by the user.

diskette drive A, B. The device identification for the diskette drives. The device identification followed by a colon (:) can precede any filename specification.

DOS. Disk Operating System.

edit. To create or change the contents of a source program or or document. For example, to insert, delete, change, rearrange, or copy lines.

Editor. A computer program that processes commands to enter lines into a source program or document, or to change them.

.EXE file. An .EXE file created by the Linker (LINK). A .EXE file of the .EXE style is an executable program. A .EXE file of the .COM style is used as input to the EXE2BIN utility for conversion to a .COM program.

expression. A source language combination of one or more arithmetic or logical operations and operands often represented by a combination of terms, possibly within parentheses.

.EXT. The optional filename extension that consists of a period and from 1 to 3 characters.

field. In a record, a specific area used for a particular piece of data.

filename. A filename consists of 1 to 8 characters. It may be immediately followed by a filename extension. See your DOS book for characters allowed in filename.

filespec. A three-part file specification consisting of an optional diskette drive ID, a required filename, and an optional filename extension.

half-byte. Either the first or last 4 consecutive bits of a byte. A hexadecimal character represents a half-byte.

hex. Hexadecimal.

hexadecimal code. A code for representing, with one digit, each value in the range of zero to fifteen:

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hexadecimal	0	1	2	3	4	5	6	7	8	9	0A	0B	0C	0D	0E	0F

Decimal	15	255	4095	65535
Hexadecimal	0F	0FF	0FFF	0FFFF

Hexadecimal is based on the radix 16.

include file. An assembler *source* file that gets imbedded into another assembler *source* file using the INCLUDE pseudo-op statement.

interface. Used by the program, module, or other unit as a description of how the implementation is structured.

instruction. A statement that specifies an operation and the values or locations of its operands if any.

.LIB. A filename extension that identifies the library file.

LIB utility. A library manager utility for building and maintaining library files.

library file. A file that contains a collection of object modules indexed (at least) by module names.

Linker (LINK). A utility used to create one load module from one or more independently translated object modules by resolving cross references among the modules.

listing file. A file, which can be printed or displayed on the screen (monitor) and which lists the source code statements, generated machine instructions, cross references, and errors.

load module. An .EXE file created by the Linker (LINK). A .EXE file of the .EXE style is an executable program. A .EXE file of the .COM style is used as input to the EXE2BIN utility for conversion to a .COM program.

loop. A set of instructions that are executed a number of times.

.LST. A filename extension that identifies a printable listing file.

machine code. A binary representation of an assembler instruction. A computer instruction written as a sequence of 0's and 1's that characterize the assembler instruction.

macro assembler. A computer program that translates a *source* program containing macro pseudo-op statements and advanced function instructions into *object* code (machine code).

mnemonic. An operation code consisting of symbols that show the operation to be performed, the type of data used, or the format of the instruction carrying out the operation.

module. A fundamental unit of code. For example: An application program can be made up of one or more *source* modules each being combined together as *object* modules at link time to form one program or executable module. Each *source* module is assembled separately to produce one *object* module.

.OBJ. A filename extension that identifies the relocatable object file, produced by an assembler or compiler.

object code. Output from the assembler, which in itself is not executable machine code but is suitable for processing by the Linker to produce executable machine code.

object module. A program unit, which is the output of the assembler.

octal code. A code based on the radix 8, where the digits 0 through 7 represent the code.

offset. The least significant word of a 4-byte long pointer, a displacement from the beginning of a segment. The offset word of a long pointer is provided either directly in the instruction, in a register, or a computational addressing mode. The offset is used to address a location relative to the base of a segment.

operand. The part of the instruction that is manipulated or operated upon.

operation. A specific action (add, multiply, move, load, etc.) that a computer carries out when instructed.

overflow. That portion of the result of an operation that exceeds the capacity of the intended unit of storage.

page. Any block of storage beginning with a byte whose address is divisible by 256.

paragraph. A 16-byte block of storage beginning at an address divisible by 16.

paragraph number. A number identifying a memory segment. This is the upper 16 bits of the 20-bit physical memory address. Also called *segment base value.*

parameter. A variable given a constant value for a specified application. One of the macro assembler command fields or modifier to the assembler invocation command.

pointer. An identifier that indicates the location of an item of data.

pop. A term used for retrieving information from a stack.

pseudo-operations. Also called pseudo-ops. Having mnemonics defining a function to be carried out by the assembler but not necessarily generating machine code.

PSP. Program Segment Prefix. A control block that DOS sets up in memory to interface with an application.

push. A term used for placing information on a stack.

radix(#). Indicates whether the number represented is hexadecimal, octal, decimal, or binary.

record. In assembler language, a collection of related information that is treated as a unit with a width ranging from 1 bit to 16 bits.

recursive. Pertaining to a process in which each step makes use of the results of earlier steps, such as when a function calls itself.

routine. Part of a program, or a sequence of instructions called by a program, that may have some general or frequent use.

.SAL. A filename extension that identifies the SALUT source file.

SALUT utility. A dual function utility to format and provide structure to assembler source files.

segment. Any contiguous block of memory up to 64K bytes (physical segment); a unit of data and/or code in your macro assembler program; contents of a segment register (logical segment); also, the basic unit of relocation and linkage.

source code. A code recognized as input by the assembler, or a compiler. Assembler *source* code follows assembler construction rules and conventions to be recognized by the assembler.

source module. The source statements or codes that constitute the input to the assembler for translation.

stack. An area in storage that stores temporary register information and returns addresses of subroutines.

stack pointer. A register that provides the current location of the stack.

string. A sequence of characters.

symbol. A method of referring to a resource of the computer; a representation of something by relationship, association, or convention.

trace. A record of the execution of a computer program; it exhibits the sequences that the instructions were executed. To record a series of events as they occur.

truncate. To delete or omit a trailing portion of a string of items.

variable. A data item that can assume any of a given set of values.

variable symbol. A symbol that can assume any of a given set of values.

word. Sixteen bits (2 bytes); enough storage for 2 characters.

Index

Special Characters

+, operator, LIB 7-7
.ASM filename extension 1-17
.BAT file 5-17
.COM file 3-17, D-23
.COM file structure D-4
.COM filename extension 1-19
.COM structured file 3-20
.CRF filename extension 1-18
.EXE file 3-7, D-23
.EXE filename extension 1-19
.EXE structured file 3-16
.LIB filename extension 1-18
.LST filename extension 1-18
.MAP filename extension 1-19
.OBJ filename extension 1-18
.REF filename extension 1-18
.SAL filename extension 1-17
.TYPE operator 2-23
+ (plus sign)
    in automatic response file 8-20
    in response to linker prompt 8-9
$DO Structure statement 4-4
$ELSE Structure statement 4-4
$ENDDO Structure statement 4-4
$ENDIF Structure statement 4-4
$ENDLOOP Structure statement 4-4
$ENDSEARCH Structure statement 4-4
$EXITIF Structure statement 4-4
$IF Structure statement 4-4
$LEAVE Structure statement 4-4
$ORELSE Structure statement 4-4
$SEARCH Structure statement 4-4
$STRTDO Structure statement 4-4
$STRTSRCH Structure statement 4-4
@ symbol, LINK

A

absolute items
    (=) pseudo-op 2-8
    constants 2-6
    EQU pseudo-op 2-8
    equates 2-8
absolute segment address
    how to determine 8-25
addition operator 2-26
addressing modes, register stack 10-9
aids, assembler source coding D-23
AND operator 2-29
application
    complex 5-20
    executing 5-22
    loading 5-22

program 5-22
running 5-22
application development 1-16
application, creating an D-23
architecture, 80286 9-3
arithmetic operators 2-26
ASCII 2-7
ASCII - 8087, conversion 10-39
ASM.EXE file 1-11
assembler 5-8, 8-4
assembler files, to see 5-18
assembler parameters
  See parameters
assembler performance 1-6
assembler source coding aids D-23
assembler work area 1-5
assembler, TRYCOM sample sessions D-4
assembly code, structuring 4-3
assembly language
  format 2-3
ASSUME pseudo-op, use of 2-41, 3-12
asterisk (*) operator, LIB 7-7
attribute
  class 8-7
  FAR type 2-5
  group 8-6
  NEAR type 2-5
  offset 2-4, 2-5
  operators 2-18
  segment 2-5, 8-6
  type 2-4
automatic response file
  linker 8-20
auxiliary carry flag 2-42
auxiliary data segment 2-41

B

backing up diskette 1-14
Base Pointers 2-40
base register, use of 2-12
base registers 2-41
base value, segment 2-41, 3-4
batch command 5-17
  example 5-17
batch file 5-17, D-10
bias, exponent 10-28, 10-29, 10-30, 10-35
binary constant 2-6
binary integer format, 8087
  long integer 10-25
  short integer 10-25
  word integer 10-25
binary integers, 8087 10-24
binary point, moving 10-28
binary real format 10-33
binary real format, Microsoft 10-35
boundary, paragraph 8-6
Break (Break) key 5-8
busy field, status word 10-11
byte isolation operators 2-20
BYTE operator 2-22
bytes, reversed order 5-23

C

carry flag 2-42
character string 2-7
characteristic, real number 10-27
CHDIR command 1-14
class attribute 8-7

co-processing, 8087 and 8088 10-45
co-processor option 10-3
code relative items 2-4
  labels 2-4
Code Segment 3-11
code, source 1-15
coding aids for source D-23
coding immediate operands 2-15
coding indexed operands 2-14
coding operands 2-10
coding storage operands 2-14
column setting 4-36
column setting rules, (SALUT) 4-37
COM file 3-17
COM file structure D-4
combined operators, LIB 7-7
command
  CHDIR 1-14
  COPY 1-10, 1-15
  DEBUG 5-23
  DIR 5-18
  DISKCOPY 1-10
  ERASE 5-19
  EXE2BIN 5-22
  MKDIR 1-14
  TYPE 5-18
command file 3-17
command line
  assembler 5-14
  CREF 6-6
  how to use, MASM 5-14
  Library Manager 7-5
  linker 8-18
  parts, MASM 5-14
command line components, MASM
  cross-ref 5-14
  list 5-14
  object 5-14
  parm 5-14
  source 5-14
command line responses
  CREF 6-6
  examples, CREF 6-6
  examples, MASM 5-14
  examples, SALUT 4-34
  Library Manager 7-8
  SALUT 4-33
command line responses, assembler 5-14
command line, SALUT
  See starting SALUT
command prompts 5-14
  assembler 5-14
  CREF 6-5
  LINK 8-7
  SALUT 4-32
commands, LIB
  entering 7-8
  response file 7-10
comment 2-32
comment field 2-32
comments, SALUT 4-6
complex application 5-20
complex operations, LIB 7-7
condition code, status word 10-11
condition, Structure statement 4-4
conditions, structure statement 4-5
console responses
  assembler 5-12
  CREF 6-5
  Library Manager 7-8
  LINK 8-17
  SALUT 4-32
constant
  binary 2-6
  character 2-7
  decimal 2-6, 2-7
  hexadecimal 2-7
  Octal 2-7

contents, diskette 1-10
Control (Ctrl) key 5-7, 5-18
control unit, 8087 10-5
control word, 8087 10-15
  (high-byte) 10-15
  (low-byte) 10-16
  infinity control (IC) 10-15
  interrupt-enable mask 10-16
  precision control (PC) 10-16
  rounding control (RC) 10-15
convention rules v
conventions, register code 2-39
conversion routines, data 10-38
conversion, data format 10-37
converting run files 5-22
converting source files 1-16
COPY command 1-10, 1-15
creating an application D-23
creating cross reference 5-19
creating load module 5-20
creating source files 1-23
CREF.EXE file 1-11
CREF program 1-18
  See also utilities
CREF termination 6-7
CREF utility 1-18
  See also Cross Reference Utility (CREF)
cross reference 6-3
  creating 5-19, 6-3
cross reference listing 5-18
cross reference program
  See utilities
Cross Reference Utility (CREF)
  Option 1 6-4
  Option 2 6-6
  starting 6-3
  stopping 6-3
  using 6-3
cross-reference listing, LIB 7-10
CRT (screen) ID 5-6
CS register, use of 3-12
Ctrl-Break 8-8
current code segment 2-41
current data segment 2-41
current stack segment 2-41

D

data conversion routines 10-38
data formats, 8087
  binary integers 10-24
  Binary real 10-33
  conversion 10-37, 10-38
  decimal integers 10-26
  long real 10-34
  Microsoft Binary real format 10-35
  real number 10-33
  real numbers 10-27
  requirements 10-33
  short real 10-33
  special values 10-31
  8087 format 10-33
data relative items 2-4
Data segment 3-9
data transfer
  direct 2-12
  immediate-to-register 2-11, 2-12
  indirect 2-12
  memory-to-register 2-12
  memory-to-register (indirect) 2-12
  register-to-memory 2-11
  register-to-register 2-11

- strings 2-11
- data transferring operands 2-10
- data types, 8087
  - addressing 10-22
  - conversion 10-38
  - long integer 10-22
  - long real 10-22
  - packed decimal 10-22
  - selecting 10-23
  - short integer 10-22
  - short real 10-22
  - temporary real 10-22
  - word integer 10-22
- data, environment 10-5
- DEBUG command 5-23
- Debug Facility, Professional 5-25
- DEBUG utility 5-23
- DEBUG, use of 2-16
- decimal
  - floating-point 2-7
  - real 2-7
- decimal constant 2-6
- decimal format, packed 10-26
- decimal integers, 8087 10-26
- decimal scientific notation 2-7
- default diskette drive 5-9, 8-8
- default extensions
  - See filename extensions
- default path 5-9, 8-8
- denormalized exception condition, 8087 10-13
- destination operand 2-10
- destination operand, 8087 10-10
- detecting exceptions 8087, methods 10-18
- development process D-2
- device identifications 5-6
  - CRT (screen) 5-6
  - diskette drives 5-6
  - LINE (non-buffered) 5-6
  - line printer 5-6
  - USER (non-buffered) 5-6
- DGROUP 8-14
- DIR command 5-18
- direct data transference 2-12
- direction flag 2-43
- DISKCOPY command 1-10
- diskette contents 1-10
- diskette drive ID 5-6
- diskette drive, default 5-9, 8-8
- Diskette files
  - ASM.EXE 1-11
  - CREF.EXE 1-11
  - FLINS.ASM 1-11
  - IBMASM.ASM 1-11
  - IBMUTIL.LIB 1-11
  - LIB.EXE 1-11
  - LINK.EXE 1-11
  - MASM.EXE 1-11
  - SALEX.SAL 1-12
  - SALEXINC.ASM 1-12
  - SALEX1.SAL 1-12
  - SALEX2.SAL 1-12
  - SALEX3.SAL 1-12
  - SALEX4.SAL 1-12
  - SALUT.EXE 1-12
  - SAMPLE2.BAT 1-12
  - SKELCOM.ASM 1-12
  - SKELEXE.ASM 1-13
  - SKELEXE2.ASM 1-13
  - STACK.ASM 1-13
  - TRYCOM1M.ASM 1-13
  - TRYCOM1S.ASM 1-13
  - TRYCOM2M.ASM 1-13
  - TRYCOM2S.ASM 1-13
  - UTEST.PAS 1-13
- diskette, backup procedure 1-14
- diskette, macro assembler samples D-10
- diskette, SALUT samples D-14

diskette, small assembler samples D-5
displacement, IP or SEG 5-24
displaying memory 5-23
DO COMPLEX Alternate, Structure statement 4-14
DO COMPLEX Structure statement 4-13
DO Structure statement 4-4
DO UNTIL Structure statement 4-12
DO WHILE Structure statement 4-12
DOS linkage examples D-23
double precision data 10-35
double precision real format 10-35
DS register, use of 3-13
DSALLOCATION 8-14
/DSALLOCATION linker parameter
DUP 1-5
DWORD operator 2-22

E

editing source code 5-19
editor
    EDLIN 1-23
    Personal Editor 1-24
EDLIN utility 1-23
eliminating errors 5-20
ELSE Structure statement 4-4, 4-11
END pseudo-op, use of 3-15, 3-17
ENDDO Structure statement 4-4
ENDIF Structure statement 4-4, 4-11
endless loop 1-8, 10-48
endless wait condition 10-49
ENDLOOP Structure statement 4-4
ENDSRCH Structure statement 4-4
enhancements, Macro Assembler 1-3
entering commands, LIB 7-8
environment data, 8087
    control word 10-5
    exception pointers 10-8
    format 10-6
    instruction pointer 10-5
    operand pointer 10-5
    status word 10-5, 10-11
    tag word 10-5
equal (EQ) operator 2-28
equate, as name field entry 2-8
ERASE command 5-19
error checking, Macro Assembler 1-3
error messages
    i/o handler A-12
    library manager A-15
    link A-18
    Macro Assembler A-3
    runtime A-14
    SALUT 4-38, A-24
    SALUTERR A-24
    Small Assembler A-3
errors
    assembly 5-17
    correcting 5-20
    displaying 5-17
    eliminating 5-20
    locating 5-23
    logical 5-20
    messages 5-17
    syntactical 5-20
errors, phase 5-5
ES register, use of 3-13

ESCAPE instruction, use of 10-45
escape instructions, co-processor 10-4
examples, logic flow Structures 4-10
examples, process blocks 4-10
examples, 8087 source D-24
exception flags, status word 10-11
exception flags, 8087 10-13
exception handler, writing 10-19
exception mask fields
  denormalized operand mask (DM) 10-16
  invalid operation mask (IM) 10-16
  overflow mask (OM) 10-16
  precision mask (PM) 10-16
  underflow mask (UM) 10-16
  Zerodivide mask (ZM) 10-16
exception pointers, 8087
  format 10-8
exceptions, 8087 10-18
  masked response 10-17
  user-written exception handler 10-19
EXE file 3-7
EXE file structure D-13
EXE filename extension 8-10
executable file 3-7
executing program 5-22
execution file 1-16
EXE2BIN command 1-17, 5-22
EXE2BIN program 1-17, 1-19
EXE2BIN utility 1-17, 1-19
EXITIF Structure statement 4-4
exponent 10-34
exponent bias 10-28, 10-29, 10-30, 10-35
exponent field 10-27
exponents, storing 10-28
expression, evaluation of 2-31
expression, operand field 2-10
extension, filename
  See filename extensions
extensions, default
  See filename extensions
extensions, filename
  See filename extensions

F

facility, Professional Debug 5-25
FAR type attribute 2-5
FAR type attribute, usage of 2-6
FAR type attribute, use of 3-12
field
  comment 2-32
  entry 2-3
  name 2-3
  operand 2-10
  operation 2-9
fields modified by LINK or LOADER 5-25
file maintenance tasks 7-3
file structure (SALUT) D-13
file types
  .BAT 5-17
  .COM 3-6, 3-17
  .EXE 3-6, 3-7
  batch 5-17
  command 3-17
  executable 3-7

file, batch D-10
filename extensions
  .ASM 1-17
  .BIN
    See DOS manual under EXE2BIN
  .COM 1-19, D-23
  .CRF 1-18
  .EXE 1-19, D-23
  .LIB 1-18, 10-38
  .LST 1-18
  .MAP 1-19
  .OBJ 1-18
  .PAS 10-38
  .REF 1-18
  .SAL 1-17
  .EXE 8-10
  .MAP 8-11
  .OBJ 8-9
  converting 5-22
  overriding 5-9
  overriding, linker 8-8
filename, NULL 5-9
files, Diskette 1-11
FINIT instruction 10-45
Flags
  auxiliary carry flag 2-42
  carry flag 2-42
  direction flag 2-43
  flag word 2-42
  interrupt flag 2-43
  overflow flag 2-43
  parity flag 2-42
  processor control 2-43
  register 2-42
  sign flag 2-43
  status 2-42
  trace flag 2-43
  zero flag 2-42
  8087 exception flags 10-11, 10-13, 10-14, 10-19
flags register 2-42
FLINS.ASM file 1-11
floating-point 1-5
floating-point decimal 2-7
floating-point, 8087 instructions summary B-16
flow control, Structures 4-9
flow structures, logic 4-9
FNCLEX instruction 10-48
FNENI instruction 10-47
FNINT instruction 10-48
FNSAVE instruction 10-48
Format (F) menu option, SALUT 4-36
format option, SALUT) 4-3
format override sub-fields, SALUT
    Opcode column (o) 4-36
    Operand column (r) 4-36
    Remark column (c) 4-36
    Structure indent (i) 4-36
format, assembly language 2-3
format, page size 7-5
format, 8087
  binary integer 10-25
  binary real 10-35
  data 10-24
  double precision real 10-35
  environment data 10-6
  exception pointer 10-8
  real number 10-29, 10-33
  save state 10-7
  single precision real 10-35
  temporary real 10-30
formats, comparing 10-36
formatting statements 2-38
formatting suggestions 2-38
forward reference 5-6
forward referencing 3-9, 5-3
framework examples D-23
FSAVE instruction 10-45

G

general information, Macro Assembler 1-15
general registers 2-39
greater than (GT) operator 2-28
greater than or equal (GE) operator 2-28
group attribute 8-6

H

hexadecimal constant 2-7
hexadecimal real 2-7
HIGH 8-14
HIGH linker parameter 8-6
high memory 8-14
HIGH operator 2-20
/HIGH linker parameter
high-byte control word 10-15
high-byte tag word 10-21
high-byte, status word 10-11

I

IBM Personal Computers, 80286-based 9-3
IBMASM.ASM file 1-11
IBMULTIL.LIB conversion routines 10-38
IBMUTIL.LIB file 1-11
identifications, device 5-6
    CRT (screen) 5-6
    diskette drives 5-6
    LINE (non-buffered) 5-6
    line printer 5-6
    USER (non-buffered) 5-6
identifier, segment 3-4
IEM, interrupt-enable mask 10-16, 10-47
IF Structure statement 4-4, 4-11
immediate operand 2-11
immediate operands, coding 2-15
implicit use of registers 2-43
indefinite value 10-31
indent format override, SALUT 4-36
Index Pointers 2-40
index register, use of 2-12
index registers 2-41
indexed operands, coding 2-14
indirect data transference 2-12
infinities, representing 10-31
infinity control arithmetic
    Affine mode 10-15
    projective mode 10-15
initializing 8087 processor 10-45
initiate, Macro Assembler 5-8
input files
    linker 8-4
instruction mnemonics, co-processor 10-4
Instruction Pointer 2-40
instruction pointer, 8087 10-5
instructions
    floating-point, 8087 instructions summary B-16
    format of B-1
    machine 2-9
    Macro 2-9
    names of B-1
    pseudo-op 2-9

    reference B-1
    save state, 8087 10-6
    80286 summary B-27
    8087 examples D-24
    8087 summary B-16
    8088 B-1
    8088 instruction summary B-1
- INT 20H 3-11
- integer format, binary 10-25
- integers, decimal 10-26
- interrupt 10-19
- interrupt flag 2-43
- interrupt 20H 3-11
- interrupt-enable mask 10-16
- interrupts, 8087
    - masking on or off 10-48
    - requests 10-47
- invalid exception condition 10-14
- invalid operation, handling 10-31
- IP or SEG displacement 5-24
- IP register, use of 3-12
- IR field, status word 10-11
- items, code relative 2-4
- items, data relative 2-4

J

- JMP instruction, use of 3-17

K

- keys
    - Break (Break) 5-8
    - Control (Ctrl) 5-7, 5-18
    - Number Lock (Num Lock) 5-7, 5-18
    - Print Screen (PrtSc) 5-7, 5-18

L

- labels
    - attributes of 2-5
- labels, SALUT 4-6
- language, machine 1-15
- LEAVE Structure statement 4-4
- length operator 2-24
- less than (LT) operator 2-28
- less than or equal (LE) operator 2-28
- LIB
    - See Library Manager
- LIB.EXE file 1-11
- LIB program 1-18, 7-3
    - See also utilities
- LIB utility 1-18, 7-3
- library 7-4
- library commands, examples of 7-8
- Library Manager
    - commands 7-8
    - cross-reference listing 7-10
    - file maintenance tasks 7-3
    - functions 7-4
    - module storage 7-3
    - object module library 7-3
    - operations 7-6
    - operators 7-7
    - response file 7-10
    - starting 7-5
    - terms 7-4
    - use of D-13

  using 7-5
Library Manager functions
  add object file 7-4
  delete object module 7-4
  extract object module 7-4
  replace object module 7-4
library manager program 7-3
Library Manager terms
  library file 7-4
  library manager program 7-4
  object file 7-4
  object module 7-4
library manager utility 7-3
library support 1-9
LINE 8-15
/LINE linker parameter
Line printer - LPT1 5-6
line printer ID 5-6
LINK
  See Linker (LINK) Utility
LINK.EXE file 1-11
LINK program
  See Linker (LINK) Utility
LINK utility
  See Linker (LINK) Utility
LINK version 1-6
LINK, modified fields 5-25
linkage, examples for DOS D-23
Linker (LINK) Utility
  command line 8-18
  command prompts 8-7
  DOS 5-20
  example session 8-22
  loading 5-21
  Macro Assembler 5-20
  messages 8-26
  starting 5-21, 8-17
linker files
  automatic response 8-4, 8-20
  input 8-4
  library 8-4, 8-12
  listing 8-4, 8-11
  object 8-4, 8-9
  output 8-4
  run 8-4, 8-10
linker files, to see
linker parameters 8-14
  See also parameters
linker prompts 8-9
linker support 1-9
linking modules 5-20, D-13
listing
  cross reference 5-18
  erase 5-19
  Octal 5-11
  printer 5-7
  source code 5-18
load module 8-16, 8-24
load operation, register stack 10-10
LOADER, modified fields 5-25
logic flow examples 4-10
Logic Flow Structures, developing 4-20
logic flow Structures, SALUT 4-9
logical operators 2-28
long integer data type 10-22
long integer, 8087 10-25
long real (real number) 10-29
long real data type 10-22
long real notation 10-34
LOOP Structures 4-4
loop, endless 10-48
low byte tag word 10-21
low byte, status word 10-13
LOW operator 2-20
low-byte control word, 8087 10-16

M

machine instructions 2-9
machine language 1-15
Macro Assembler
  diskette contents 1-10
  enhancements 1-3
  execution, stopping 5-8
  exit 5-17
  general information 1-15
  introduction 1-3
  memory requirements 1-7
  running 5-8
  starting 5-8
  stop 5-17
  storage requirements 1-7
  support 1-8
  technical differences 1-3
  termination 5-17
  usage suggestions/information 5-8
  using 5-8
  2-pass 5-3
macro assembler sample session D-10
Macro instructions 2-9
mantissa, real number 10-27
MAP 8-15
MAP extension 8-11
/MAP linker parameter
mask bit, resetting 10-48
mask bit, setting/resetting 10-19
mask fields, exception 10-16
MASK operator 2-25
masked response
  denormalized (D) 10-17
  invalid operation (I) 10-17
  overflow (O) 10-17
  precision (P) 10-17
  underflow (U) 10-17
  Zerodivide (Z) 10-17
masking 10-14
MASM.EXE file 1-11
math co-processor, 80287 9-6
memory
  addressing 3-3
  high 8-14
  low 8-14
  paragraph 3-3
  segment 3-4
memory map printout 5-21
memory requirements 1-7
memory, reversed order of bytes 5-23
menu options, SALUT
  Format (F) 4-36
  Pre-process (P) 4-36
  Quit (Q) 4-35
  Replace tabs (R) 4-37
messages
  i/o handler A-12
  library manager A-15
  link A-18
  Macro Assembler A-3
  runtime A-14
  SALUT A-24
  SALUTERR A-24
  Small Assembler A-3
messages, linker 8-26
Microsoft binary real format 10-33
minus (-) operator, LIB 7-7
MKDIR command 1-14
MOD 2-32
modified fields, by LINK or LOADER 5-25
modular programming practices 7-3
module
  load, composite 5-20
  load, producing 5-20
module storage 7-3
module, object 1-15, 5-18
modules, linking D-13

modulo operator 2-27
MOV instruction, use of 3-13
moving binary point 10-28
multiplication operator 2-26

N

name entries, creating 2-34
name field
    entries 2-3
    items 2-3
names of instructions B-1
names of pseudo-ops C-1
NAN (Not-A-Number) special value 10-32
NEAR type attribute 2-5
NEAR type attribute, usage of 2-6
NEAR type attribute, use of 3-17
nested segments 1-5
NMI, non-maskable interrupt 10-48
/NOD linker parameter
/NOG linker parameter
non-interactive mode, SALUT 4-33
non-normal special values 10-32
normalizing real numbers 10-27
not equal (NE) operator 2-28
NOT operator 2-30
notation, long real 10-34
notation, short real 10-34
NULL filename 5-9
Number Lock (Num Lock) key 5-7, 5-18
numbers, representing 2-6
numeric execution unit, 8087 10-5

O

OBJ extension 8-9
object file 1-16, 7-4
object file, add to LIB 7-4
object module 1-15, 5-18, 7-4
object module library 7-3
object module library files 7-4
object module, delete 7-4
object module, extract 7-4
object module, replace 7-4
object modules 8-9
Octal constant 2-7
Octal listing 5-11
offset 3-5
offset attribute 2-4, 2-5
OFFSET operator 2-21
offset register 3-3
opcode field column override, SALUT 4-36
operand field 2-10
operand field column override, SALUT 4-36
operand pointer, 8087 10-5
operands 2-10
    coding 2-10, 2-14
    data-transferring 2-10
    destination 2-10
    immediate 2-11
    register 2-11
    source 2-10
    storage 2-12
    structure fields 2-13
operation field 2-9
operation, invalid 10-31
operation, store-and-pop 10-10

operators 2-18
  .TYPE 2-23
  addition 2-26
  AND 2-29
  arithmetic 2-26
  attribute 2-18
  byte isolation 2-20
  equal (EQ) 2-28
  greater than (GT) 2-28
  greater than or equal (GE) 2-28
  HIGH 2-20
  length 2-24
  less than (LT) 2-28
  less than or equal (LE) 2-28
  logical 2-28
  LOW 2-20
  MASK 2-25
  modulo 2-27
  multiplication 2-26
  NOT 2-30
  not equal (NE) 2-28
  OFFSET 2-21
  OR 2-29
  pointer 2-18
  precedence of 2-31
  record-specific 2-25
  relational 2-28
  SEG 2-21
  segment 2-19
  shift count 2-25
  shift left 2-27
  shift right 2-27
  short 2-19
  size 2-24
  subtraction 2-26
  THIS 2-20
  type 2-22
  value-returning 2-20
  width 2-25
  XOR 2-30
operators, combined 7-7
operators, LIB
  asterisk (*) 7-7
  minus (-) 7-7
  plus (+) 7-7
option menu, SALUT 4-35
option 1, CREF 6-4
Option 1, LINK 8-17
option 1, MASM 5-12
Option 2, CREF 6-6
option 2, LINK 8-18
option 2, MASM 5-14
option 3, LINK 8-20
option 3, MASM 5-17
option, LIB page size 7-5
OR operator 2-29
order of bytes reversed 5-23
ordering of segments 1-6
ORELSE Structure statement 4-4
ORG pseudo-op, use of 3-17
output files
  linker 8-4
overflow exception condition 10-13
overflow flag 2-43

P

Packed decimal 1-4
packed decimal data type 10-22
packed decimal format, 8087 10-26
page sizes, LIB
  format 7-5
paragraph 3-3
paragraph boundary 8-6
parallel operation 10-3
parameters 5-10
  assembler 5-10

- /A 5-10
- /D 5-10
- /DS 8-14
- /E 5-10
- /HIGH 8-14
- /LINE 8-15
- /MAP 8-15
- /N 5-11
- /NOD 8-15
- /NOG 8-15
- /O 5-11
- /PAUSE 8-16
- /R 4-33, 5-11
- /S 5-11
- /STACK 8-16
- /X 5-11
- linker 8-14
- SALUT 4-33
- specifying 5-9, 8-8

parity flag 2-42
parms
  See parameters
pass 1, assembler 5-3
pass 2, assembler 5-3
path 5-9, 8-8
  default 5-9, 8-8
  specifying 5-9, 8-8
Pathname support 1-8
PAUSE 8-16
/PAUSE linker parameter
performance, assembler 1-6
Personal Editor program 5-18
Personal Editor utility 1-24, 5-18
phase error 5-3
phase error, causes 5-5
physical address 3-5
plus (+) operator, LIB 7-7
plus sign
  in automatic response file 8-20
  in response to linker prompt 8-9
pointer operator 2-18
pointer registers 2-39
POP 3-13
Pre-process (P) menu option, SALUT 4-36
precedence of operators 2-31
precision control
  long real 10-16
  reserved 10-16
  short real 10-16
  temporary real 10-16
precision exception condition 10-13
Print Screen (PrtSc) key 5-7, 5-18
printer (LPT1) 5-6
printer listing 5-7
printing
  source code 5-18
  source errors 5-18
printing symbol table 1-6
printout
  cross reference 5-19
  memory map 5-21
  source code 5-7
PROC pseudo-op, use of 3-12
process blocks 4-10
process blocks, developing 4-20
process blocks, examples 4-10
process, development D-2
processor control flags 2-43
processor control, 8087
  interrupts and waits 10-47
program
  [M]ASM 1-18
  CREF 1-18
  cross reference 1-18
  DEBUG 5-23
  EDLIN 1-23
  executing 5-22
  EXE2BIN 1-19
  large assembler 1-18

  - LIB 1-18
  - library manager 1-18
  - LINK 1-19
  - linker 1-19
  - Personal Editor 1-24, 5-18
  - Professional DEBUG Facility 5-25
  - SALUT 1-17
  - small assembler 1-18
  - structured assembler 1-17
- Program Segment Prefix (PSP) 3-11
- program structure 3-6
- program, running 5-22
- program, SALUT
  - See Structured Assembly Language Utilities
- programs, sample 1-9
- prompt mode, SALUT 4-35
- prompts 5-8, 5-14
  - format of, CREF 6-5
  - format of, LINK 8-7
  - format of, MASM 5-12
  - response to, LINK 8-18
  - responses to, CREF 6-5
  - responses to, MASM 5-12
- Protected Virtual Address Mode, 80286 and 80287 9-3
- pseudo operation 2-9
- pseudo-op 2-9
- pseudo-ops C-1
  - format C-1
  - names C-1
  - reference C-1
  - summary C-1
- PSP 3-11, 3-13, 3-17, 3-19, 3-22, 8-26
- public symbols 8-24
- PUSH pseudo-op, use of 3-13

Q

- Quit (Q) menu option, SALUT 4-35
- QWORD operator 2-22

R

- Real Address Mode, 80286 and 80287 9-3
- real decimal 2-7
- real format, temporary 10-30
- real number format 10-33
- real number format, 8087
  - description of 10-33
  - long 10-34
  - long real 10-29
  - short 10-33
  - short real 10-29
- real numbers, 8087
  - characteristic 10-27
  - exponent field 10-27
  - mantissa 10-27
  - normalizing 10-27
  - sign field 10-27
  - significand field 10-27
- real, hexadecimal 2-7
- record-specific operators 2-25
- referencing, forward 5-3
- register code conventions 2-39
- register operand 2-11
- register stack, 8087
  - addressing modes 10-9
  - exponent field 10-9
  - fields 10-9
  - load operation 10-10
  - sign field 10-9
  - significand field 10-9

  - store-and-pop operation 10-10
- register, control word 10-15
- register, offset 3-3
- registers 2-39
  - base 2-41
  - base pointer 2-40
  - flag 2-42
  - general 2-39
  - implicit use 2-43
  - index 2-41
  - index pointer 2-40
  - instruction pointer 2-40
  - pointer 2-39
  - segment 2-41
  - stack pointer 2-40
- REGMATH, libraries 10-38
- relational operators 2-28
- relative zero 8-24
- relocatable loader 8-4
- remark field column override, SALUT 4-36
- replace tabs option (SALUT) 4-3
- Replace tabs option, SALUT 4-37
- representing infinities 10-31
- representing special values 10-31
- requirements, data format 10-33
- resetting mask bit 10-19, 10-48
- response file commands, LIB 7-10
- responses to command line prompts
  - See command line responses
- RET instruction, use of 3-14
- return code 1-6
- reversed order of bytes 5-23
- rounding control
  - chop 10-15
  - round down 10-15
  - round to nearest 10-15
  - round up 10-15
- routines, conversion 10-38
- rules, convention v
- rules, statement 2-33
- run file 8-10
- run files, converting 5-22
- running program 5-22
- running SALUT 4-32

S

- SALEX.SAL file 1-12
- SALEXINC.ASM file 1-12
- SALEX1.SAL file 1-12
- SALEX2.SAL file 1-12
- SALEX3.SAL file 1-12
- SALEX4.SAL file 1-12
- SALUT
  - See Structured Assembly Language Utilities
- SALUT.EXE file 1-12
- SALUT examples D-13
- SALUT file structure D-13
- SALUT program 1-16
  - See also Structured Assembly Language Utilities
- SALUT Sample Session D-12
- SALUT sample session summary D-22
- SALUT support 1-9
- SALUT utility 1-16, 4-3
- SALUTERR A-24
- SALUTERR message 4-38
- sample programs 1-9
- sample sessions 1-9
  - batch file D-10

- batch file procedure D-10
- diskette, macro assembler samples D-10
- diskette, SALUT samples D-14
- diskette, small assembler samples D-5
- introduction D-1
- macro assembler D-10
- procedure, macro assembler D-10
- procedure, SALUT sample D-16
- procedure, small assembler D-6
- programs D-4
- requirements D-3
- SALEX examples D-12
- SALUT D-12
- SALUT examples D-13
- small assembler D-5
- source coding aids D-23
- summary, SALUT sample D-22
- summary, small assembler D-9
- TRYCOM D-4
- TRYCOM examples D-4
- using D-4
- 8087 examples D-24

SAMPLE2.BAT file 1-12
save state instructions, 8087 10-6
scientific notation, decimal 2-7
screen
- echoing 5-7, 5-18
- scrolling 5-7, 5-18

SEARCH COMPLEX Alternate, Structure statement 4-19
SEARCH COMPLEX Structure statement 4-17
SEARCH Structure statement 4-4
SEARCH UNTIL Structure statement 4-15
SEARCH WHILE Structure statement 4-16
seg ID 2-41, 3-4
SEG operator 2-21
SEG or IP displacement 5-24
segment 3-4, 8-6, 8-9
- auxiliary data 2-41
- Code 3-7, 3-11
- current code 2-41
- current data 2-41
- current stack 2-41
- Data 3-7, 3-9
- operator 2-19
- Stack 3-7

segment attribute 2-4, 2-5, 8-6
segment base value 2-41, 3-4
SEGMENT command 8-16
segment identifier 2-41, 3-4
segment names
- as name field entry 2-8

SEGMENT pseudo-op, use of 3-8
segment register 3-3
segment registers 2-41
segment registers, use of 3-11
segmentation 3-3
segments, class 8-7
segments, nesting 1-5
segments, ordering of 1-6
selecting data types, 8087 10-23
setting columns, SALUT 4-37
setting mask bit 10-19
shift count 2-25
shift operator 2-27
shift right operator 2-27
SHL 2-32
SHL, SHR operator 1-4
short integer data type 10-22

short integer, 8087 10-25
short operator 2-19
short real (real number) 10-29
short real data type 10-22
short real notation 10-34
short real number format 10-34
SHR 2-32
sign field 10-27
sign flag 2-43
significand field 10-27
single precision data 10-35
single precision real format 10-35
size operator 2-24
SKELCOM.ASM file 1-12, D-23
skeletal, source coding aids D-23
SKELEXE.ASM file 1-13, D-23
SKELEXE2.ASM file 1-13, D-23
Small Assembler
  functions not supported 1-15
  memory requirements 1-7
  using 1-15
small assembler sample session D-5
small assembler session summary D-9
source code 1-15
source code printout 5-7
source code, editing 5-19
source code, translating 1-15
source coding aids D-23
source file 1-16
source file, converting 1-16
source file, creating 1-23
source operand 2-10
special values
  NAN (Not-A-Number) 10-32
  non-normals 10-32
specifying path 5-9, 8-8
ST register, 8087 10-10
stack 3-13
STACK.ASM file 1-13
stack allocation statement 8-16
/STACK linker parameter
Stack Pointer 2-40
Stack segment 3-7
stack, 8087 register 10-10
starting assembler 5-8
  batch command 5-17
  command line responses 5-14
  console responses 5-12
  option 1 5-12
  option 2 5-14
  option 3 5-17
starting SALUT
  command line responses 4-33
  drive name, specifying 4-32
  input file, specifying 4-32
  non-interactive mode responses 4-33
  option menu 4-35
  options 4-33, 4-35
  parameters 4-33
  prompt mode responses 4-35
starting statement 3-17
starting the linker
  Option 1 8-17
  Option 2 8-18
  Option 3 8-20
statement
  assembler 2-3
  comment field 2-3
  definition 2-3
  field order 2-3

- fields 2-3
- format 2-38
- name field 2-3
- operand field 2-3
- operation field 2-3
- parts 2-3
- rules 2-33
- rules, all entries 2-33
- rules, comment field 2-37
- rules, name field 2-34
- rules, operand field 2-36
- rules, operation field 2-35

status flags 2-42
status word
- (high-byte) 10-11
- (low-byte) 10-13
- busy field 10-11
- condition code 10-11
- exception flags 10-11
- IR field 10-11
- TOP field 10-10, 10-11

status word register 10-11
status word, 8087 10-5
stopping assembler 5-17
storage operand 2-12
storage operands, coding 2-14
storage requirements 1-7
store-and-pop operation, 8087 10-10
storing exponents 10-28
STRTDO Structure statement 4-4
STRTSRCH Structure statement 4-4
STRUC 1-5
structure
structure operand 2-13
Structure statement conditions 4-5
Structure statements, SALUT
- $DO 4-4
- $ELSE 4-4
- $ENDDO 4-4
- $ENDIF 4-4
- $ENDLOOP 4-4
- $ENDSRCH 4-4
- $EXITIF 4-4
- $IF 4-4
- $LEAVE 4-4
- $ORELSE 4-4
- $SEARCH 4-4
- $STRTDO 4-4
- $STRTSRCH 4-4
- comments 4-6
- conditions 4-4
- DO COMPLEX 4-13
- DO COMPLEX, Alternate 4-14
- DO UNTIL 4-12
- DO WHILE 4-12
- ENDIF 4-11
- error messages 4-38
- how to use 4-20
- IF 4-11
- IF ELSE 4-11
- labels 4-6
- SEARCH COMPLEX 4-17
- SEARCH COMPLEX, Alternate 4-19
- SEARCH UNTIL 4-15
- SEARCH WHILE 4-16
- syntax 4-6

Structured Assembly Language Utilities
- error messages 4-37
- flow control structures 4-9
- format option 4-3
- format override sub-fields 4-36
- how to start 4-32
- logic flow Structures 4-9
- LOOPS 4-4
- primary function 4-3
- process blocks 4-10
- purpose 4-3

  - replace tabs option 4-3
  - secondary functions 4-3
  - Structure statement conditions 4-5
  - Structure statements 4-4
  - uses 4-3
  - using 4-20
- structured file, .COM 3-20
- structured file, .EXE 3-16
- structured programming foundation 7-3
- structures 4-3
- Structures, logic flow 4-9
- structuring assembly code 4-3
- sub-directory 1-14
- subtraction operator 2-26
- summary of
  - pseudo-ops C-1
  - 80286 instructions B-27
  - 8087 instructions B-16
  - 8088 instructions B-1
- switch
  - See parameters
- symbol table, printing 1-6
- symbols, global and public 8-15
- syntax requirements
  - comment field 2-37
  - immediate operands 2-15
  - name field 2-34
  - operand field 2-36
  - operation field 2-35
  - storage operands 2-14
- syntax rules, column setting 4-37
- syntax rules, statement 2-33
- syntax, SALUT
  - AND/OR, using 4-6
  - comments, using 4-6
  - invalid conditional 4-7
  - labels, using 4-6
  - LONG, using 4-6
  - NCXZ/CXZ, using 4-7
  - NUL, using 4-6
- syntax, Structure statements 4-6

T

- tag word
  - (high-byte) 10-21
  - (low-byte) 10-21
- tag word register 10-21
- tag word, 8087 10-5
- TBYTE operator 2-22
- technical differences, Macro Assembler 1-3
- temporary file, VM.TMP
- temporary real data type 10-22
- temporary real format, 8087
- terminating error messages, SALUT 4-38
- termination
  - assembler 5-17
  - CREF 6-7
- THIS operator 2-20
- TOP field, status word 10-11
- trace flag 2-43
- transferring operands, data 2-10
- translating source code 1-15
- TRYCOM examples D-4
- TRYCOM1M.ASM file 1-13
- TRYCOM1S.ASM file 1-13
- TRYCOM2M.ASM file 1-13
- TRYCOM2S.ASM file 1-13
- type attribute 2-4
- TYPE command 5-18
- TYPE mismatch 1-3
- type operator 2-22
  - BYTE 2-22
  - DWORD 2-22
  - QWORD 2-22

- TBYTE 2-22
- WORD 2-22

U

UNASSEMBLE command, DEBUG 2-16
underflow exception condition 10-13
unending wait 8087, preventing 10-49
using assembler 5-8
UTEST.PAS file 1-13
UTEST.PAS conversion file 10-38
utilities 1-17
  [M]ASM 1-18
  CREF 1-18, 6-3
  cross reference 1-18, 6-3
  DEBUG 2-16, 5-23
  EDLIN 1-23
  EXE2BIN 1-19
  large assembler 1-18
  LIB 1-18, 7-3
  library manager 1-18, 7-3
  LINK 1-19
  linker 1-19
  Personal Editor 1-24, 5-18
  SALUT 1-17, 4-3
  small assembler 1-18
  structured assembler 1-17
utility, Professional Debug 5-25

V

value-returning operators 2-20
value, indefinite 10-31
variable
vector 3-5, 3-13
version 1.0, Macro Assembler 1-3
version 2.0, Macro Assembler 1-3
VM.TMP temporary file

W

WAIT instruction 10-45
waits, 8087
  endless 10-49
  unending 10-49
width operator 2-25
word integer data type 10-22
word integer, 8087 10-25
WORD operator 2-22
word, register flag 2-42
work area, assembler 1-5

X

XOR operator 1-4, 2-30

Z

zero flag 2-42
Zerodivide exception condition 10-13

Numerals

80286 instructions summary B-27
80286 support 1-8
80286, architecture 9-3
80286, Protected Virtual Address Mode 9-3
80286, Real Address Mode 9-3
80287 math co-processor 9-6
80287, Protected Virtual Address Mode 9-3
80287, Real Address Mode 9-3
8087 - ASCII, conversion 10-39
8087 architecture
  control unit 10-5
  numeric execution unit 10-5
8087 assembler sample source D-24
8087 data format 10-33
8087 exception conditions
  denormalized (D) 10-13
  invalid (I) 10-14
  masking 10-14
  overflow (O) 10-13
  precision (P) 10-13
  underflow (U) 10-13
  Zerodivide (Z) 10-13
8087 instructions summary B-16
8087 Math Co-Processor
  architecture 10-5
  escape instructions 10-4
  following initialization 10-46
  initializing 10-45
  instruction mnemonics 10-4
  introduction 10-3
  register stack operation 10-10
8087/80287 support 1-8
8087, data format 10-38
8088 instructions summary B-1

IBM

The Personal Computer
Computer Language
Series

Reader's Comment Form

Macro Assembler 6361612

Your comments assist us in improving the usefulness of our publication; they are an important part of the input used for revisions.

IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Please do not use this form for technical questions regarding the IBM Personal Computer or programs for the IBM Personal Computer, or for requests for additional publications; this only delays the response. Instead, direct your inquiries or request to your authorized IBM Personal Computer dealer.

Comments:

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 321 BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

Fold here

Tape Please do not staple Tape

The Personal Computer
Computer Language
Series

Reader's Comment Form

Macro Assembler 6361612

Your comments assist us in improving the usefulness of our publication; they are an important part of the input used for revisions.

IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Please do not use this form for technical questions regarding the IBM Personal Computer or programs for the IBM Personal Computer, or for requests for additional publications; this only delays the response. Instead, direct your inquiries or request to your authorized IBM Personal Computer dealer.

Comments:

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 321 BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

Fold here

Tape Please do not staple Tape

© IBM Corp. 1984
All rights reserved.

International Business
Machines Corporation
P.O. Box 1328-S
Boca Raton, Florida 33432

Printed in the
United States of America

6361612